NOTICE

Apple Computer Inc. reserves the right to make improvements in the product described in this manual at any time and without notice.

DISCLAIMER OF ALL WARRANTIES AND LIABILITY

APPLE COMPUTER INC. MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL OR WITH RESPECT TO THE SOFTWARE DESCRIBED IN THIS MANUAL, ITS QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. APPLE COMPUTER INC. SOFTWARE IS SOLD OR LICENSED "AS IS". THE ENTIRE RISK AS TO ITS QUALITY AND PERFORMANCE IS WITH THE BUYER. SHOULD THE PROGRAMS PROVE DEFECTIVE FOLLOWING THEIR PURCHASE, THE BUYER (AND NOT APPLE COMPUTER INC., ITS DISTRIBUTOR, OR ITS RETAILER) ASSUMES THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR, OR CORRECTION AND ANY INCIDENTAL OR CONSEQUENTIAL DAMAGES. IN NO EVENT WILL APPLE COMPUTER INC. BE LIABLE FOR DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT IN THE SOFTWARE, EVEN IF APPLE COMPUTER INC. HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF IMPLIED WARRANTIES OR LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

1Ø26Ø Bandley Drive
Cupertino, California 95Ø14
(4Ø8) 996-1Ø1Ø

APPLE Product #A2LØØ27
(Ø3Ø-Ø1Ø1-ØØ)

Apple II

Apple Pascal

Language Reference Manual

ACKNOWLEDGEMENTS

The Apple Pascal™ system incorporates UCSD Pascal™ and Apple extensions for graphics and other functions. UCSD Pascal was developed largely by the Institute for Information Science at the University of California at San Diego, under the direction of Kenneth L. Bowles.

TABLE OF CONTENTS

APPLE PASCAL LANGUAGE

CHAPTER 3

BUILT-IN PROCEDURES AND FUNCTIONS

CHAPTER 4

THE PASCAL COMPILER 57

CHAPTER 5

PROGRAMS IN PIECES 71

CHAPTER 6

OTHER DIFFERENCES 83

CHAPTER 7

SPECIAL UNITS FOR THE APPLE 89

APPENDIX A

DEMONSTRATION PROGRAMS 107

APPENDIX B

TABLES 131

APPENDIX C

ADDITIONAL TEXT I/O DETAILS 143

APPLE PASCAL LANGUAGE

APPENDIX D

ONE-DRIVE STARTUP 147

APPENDIX E

TWO-DRIVE STARTUP 169

APPENDIX F

APPLE PASCAL SYNTAX 191

INDEX 205

APPLE PASCAL LANGUAGE

CHAPTER 1
INTRODUCTION

GETTING STARTED

If you don't already know how to start up the Apple Pascal Operating System for use with the Apple Pascal language, please read Appendix D if you have one diskette drive, or Appendix E if you have two or more diskette drives. Each of these Appendices is a tutorial session, covering system startup, diskette initialization, diskette copying, and a demonstration of Apple Pascal programming.

SCOPE OF THIS DOCUMENT

This document covers the features of the Apple Pascal programming language that are different from the "Standard Pascal" language defined by Jensen and Wirth in the Pascal User Manual and Report (Springer-Verlag, New York, 1978). This includes the differences introduced in UCSD Pascal, and also special extensions of UCSD Pascal for the Apple computer.

The Apple Pascal system facilities such as the Editor, the Linker, etc. are covered in the Apple Pascal Operating System Reference Manual. These facilities are useful in various applications besides Apple Pascal programming; they are discussed here only as they relate specifically to Apple Pascal programs.

HOW TO USE THIS DOCUMENT

To use this document you must either have a thorough knowledge of Standard or UCSD Pascal, or use some book or manual that fully describes Standard or UCSD Pascal. This is a reference manual, designed to give you the facts without very much emphasis on teaching you Pascal.

You should also have the Apple Pascal Operating System Reference Manual, which gives complete information on the various system facilities that support the creation and development of Apple Pascal programs.

One aspect of the Apple Pascal Operating System is covered in this manual: the procedures for starting up the system when your purpose is to work with Apple Pascal programs. Appendices D and E describe these procedures.

At various places in the text you will see the special symbol

which indicates a feature that you need to be cautious about. Another special symbol is

which indicates a particularly useful piece of information (usually something that is not obvious).

ORGANIZATION

Chapters 2 and 3 cover the large differences in Apple Pascal that will have the most immediate programming impact: the differences in predefined types, procedures, and functions, especially the procedures for input and output.

Chapter 4 covers the compiler operation and the compiler options, which are powerful and important. Further details on compiler operation can be found in the Apple Pascal Operating System Reference Manual.

Chapter 5 covers techniques for breaking a program into separate pieces which can be linked together. These techniques are another major area of difference but are not needed for small programs.

Chapter 6 gives the remaining differences in the language, which are of minor impact for most programs.

Chapter 7 covers the extremely powerful library options of Apple Pascal, including the Turtlegraphics package.

Appendix A presents a fully annotated program that uses graphics, and also describes the demonstration programs supplied with Apple Pascal.

Appendix B contains various tables relating to the Apple Pascal Language and the system.

Appendix C gives some technical details on textfile I/O operations.

Appendices D and E cover system startup and essential operating procedures for use with the Apple Pascal language.

Appendix F is a complete set of syntax diagrams for the Apple Pascal language.

NOTATION USED IN THIS MANUAL

In syntax descriptions, the following convention is used:

- Square brackets [] are used to enclose anything that may legally be omitted from the syntax.

DIFFERENCES BETWEEN APPLE AND STANDARD PASCAL

The major differences are summarized below; see Chapter 6 for the minor ones.

PREDEFINED VARIABLE TYPES

- A new variable type, STRING, supported by a set of new built-in procedures and functions. See Chapters 2 and 3.
- A new file type, INTERACTIVE, supported by the extended file I/O procedures and functions. See Chapters 2 and 3.
- Minor restrictions on SET types.
- Minor differences in the treatment of PACKED variables. Automatic PACK and UNPACK operations, with elimination of the PACK and UNPACK procedures of Standard Pascal. See Chapter 2.
- An extension of the INTEGER type called LONG INTEGER. A LONG INTEGER is a value represented by up to 36 binary-coded decimal (BCD) digits. See Chapter 2.

BUILT-IN PROCEDURES AND FUNCTIONS

These are the procedures and functions that are part of the Apple Pascal language itself, as opposed to special-purpose functions implemented in the system library. Built-in procedures and functions are called "built-ins" for short.

- New built-ins supporting STRING variables. See Chapters 2 and 3.
- Extended definitions of the built-ins for file I/O, supporting INTERACTIVE files. See Chapters 2 and 3.
- A set of new byte-oriented built-ins. See Chapter 3.
- New built-ins called MARK and RELEASE which replace the DISPOSE of Standard Pascal. See Chapter 3.
- Other new built-ins and redefinitions of Standard Pascal built-ins. See Chapter 3.
- The transcendental functions SIN, COS, EXP, ATAN, LN, LOG, and SQRT are not built-ins in Apple Pascal. They are provided as library functions. See Chapter 7.

BREAKING PROGRAMS INTO PIECES

- SEGMENT procedures and functions, which reside in memory only when active. See Chapter 5.
- UNITS, which are separately compiled collections of procedures that can be integrated into any host program via a library facility. See Chapter 5.
- EXTERNAL procedures and functions, which are declared in an Apple Pascal program but implemented in assembly language and then integrated into a host program via the library facility. See Chapter 5.

SPECIAL UNITS FOR THE APPLE

- These are major facilities for the Apple, implemented as UNITs in a system library. They include the Turtlegraphics package for the high-resolution color display of the Apple. See Chapter 7.

CHAPTER 2
PREDEFINED TYPES

In addition to the predefined types of Standard Pascal (REAL, INTEGER, CHAR, ARRAY, etc.), Apple Pascal has a STRING type, an INTERACTIVE file type, and a LONG INTEGER type.

Also, the details of certain other predefined types differ from Standard Pascal.

THE STRING TYPE

Apple Pascal has a new predeclared type, STRING. The value of a STRING variable is a sequence of characters. Variables of type STRING are essentially PACKED ARRAYs OF CHAR that have a dynamically changing number of elements (characters). However, the value of a STRING variable cannot be assigned to a PACKED ARRAY OF CHAR, and the value of a PACKED ARRAY OF CHAR cannot be assigned to a STRING variable. Strings are supported by a set of built-in procedures and functions; see Chapter 3.

The number of characters in a string at any moment is the length of the string. The default maximum length of a STRING variable is 80 characters, but this can be overridden in the declaration of a STRING variable (up to the absolute limit of 255). To do so, put the desired maximum length in [brackets] after the type identifier STRING. Examples of declarations of STRING variables are:

```
TITLE: STRING;  (* defaults to a maximum length of 80 characters *)

NAME: STRING[30];  (* allows the STRING to be a maximum of 30
                      characters*)
```

The value of a STRING variable can be altered by using an assignment statement with a string constant or another STRING variable:

```
TITLE := '    THIS IS A TITLE    '

  or

NAME := TITLE
```

or by means of the READ procedure as described in the next chapter:

```
READLN(TITLE)
```

or by means of the STRING built-ins, also described in the next chapter:

```
NAME:= COPY(TITLE,1,30)
```

Note that a string constant may not contain an end-of-line; the constant must be on a single line in the program.

The individual characters within a STRING are indexed from 1 to the LENGTH of the STRING. LENGTH is a built-in function which is described in Chapter 3. For example, if TITLE is the name of a string, then

```
TITLE[1]
```

is a reference to the first character of TITLE, and

```
TITLE[ LENGTH(TITLE) ]
```

is a reference to the last character of TITLE.

A variable of type STRING may be compared to any other variable of type STRING or to a string constant, regardless of its current dynamic length. The comparison is lexicographical: i.e., one string is "greater than" another if it would come first in an alphabetic list of strings. The ordering of the ASCII character set (see Appendix B) is used to determine this. The following program is a demonstration of legal comparisons involving variables of type STRING:

```
PROGRAM  COMPARESTRINGS;
VAR S: STRING;
    T: STRING[40];

BEGIN
  S:= 'SOMETHING';
  T:= 'SOMETHING BIGGER';
  IF S = T THEN
    WRITELN('Strings do not work very well')
  ELSE
    IF S > T THEN
      WRITELN(S,' is greater than ',T)
    ELSE
      IF S < T THEN
        WRITELN(S,' is less than ',T);
  IF S = 'SOMETHING' THEN
    WRITELN(S,'  equals  ',S);
  IF S > 'SAMETHING' THEN
    WRITELN(S,' is greater than SAMETHING');
  IF S = 'SOMETHING                ' THEN
    WRITELN('BLANKS DON''T COUNT')
  ELSE
    WRITELN('BLANKS APPEAR TO MAKE A DIFFERENCE');
  S:='XXX';
  T:='ABCDEF';
  IF S > T THEN
    WRITELN(S,' is greater than ',T)
  ELSE
    WRITELN(S,' is less than ',T)
END.
```

The above program produces the following output:

```
SOMETHING is less than SOMETHING BIGGER
SOMETHING  equals  SOMETHING
SOMETHING is greater than SAMETHING
BLANKS APPEAR TO MAKE A DIFEERENCE
XXX is greater than ABCDEF
```

Strings can also be declared as constants, as in the following:

```
PROGRAM BAZ;

CONST SAMMY = 'Hi there, I''m Sammy the String!';

BEGIN
  WRITELN(SAMMY)
END.
```

The use of STRING variables is discussed further in the next chapter, in connection with the built-in procedures and functions of Apple Pascal.

A variable of type STRING cannot be indexed beyond its current dynamic length. The following sequence will result in an invalid index run-time error:

```
TITLE:= '1234';
TITLE[5]:= '5'
```

Beware of zero-length strings: they cannot be indexed at all without causing unpredictable results or a run-time error. If a program indexes a string that might have zero length, it should first use the LENGTH function to see if the length is zero. If the length is zero, the program should not execute statements that index the string. See Chapter 3 for details on the LENGTH function.

Notice that a string value containing only one character is not the same thing as a CHAR value; strings and CHARs are distinct data types. The one exception is that a string constant containing only one character has exactly the same form as a CHAR constant, and such a constant can be used as either a CHAR value or a string value.

You cannot define a function of type STRING. However, there are built-in functions of type STRING as described in the next chapter.

THE FILE TYPES

A NOTE ON TERMINOLOGY

For every file named F that is declared in a Pascal program, there is an automatically declared variable named F^. This is the "buffer variable" of the file. Some Pascal manuals also use the looser term "window" to describe the way that different file records can be loaded into the buffer variable. This manual, instead, talks about a "file pointer" associated with each open file. The file pointer points to one record in the file, which is called the "current record." Please understand that the file pointer is not a Pascal POINTER variable but just a convenient way of discussing file records.

The following sections describe Apple Pascal's special file features: the INTERACTIVE file type, untyped files, predefined files, and a special format for files of characters.

INTERACTIVE FILES

Like a TEXT file, an INTERACTIVE file is a file of characters. The difference is in the way INTERACTIVE and TEXT files are handled by the RESET, READ, and READLN procedures.

When a Pascal program READs characters from a TEXT file, the program must first open the file with RESET. RESET automatically performs a GET operation: that is, it loads the first character of the file into the file's buffer variable and then advances the file pointer to the next character. A subsequent READ or READLN with a variable of type CHAR begins its operation by first taking the character that is already in the buffer variable and then performing a GET.

If the file is of type INTERACTIVE instead of TEXT, the opening RESET does <u>not</u> perform a GET. The buffer variable is undefined and the file pointer points to the first character of the file instead of the second. Therefore, a subsequent READ or READLN has to begin its operation by first performing a GET and then taking the character that was placed in the buffer variable by the GET. This is the reverse of the READ sequence used with a TEXT file.

There is one primary reason for using the INTERACTIVE type. If a file is not a diskette file but represents a device such as the keyboard, it is not possible to perform a GET on it until a character has been typed. If RESET tried to do a GET, the program would then go no further until a character was typed. With the INTERACTIVE type, the program doesn't perform a GET until it is executing a READ or READLN. The standard predeclared files INPUT and OUTPUT are INTERACTIVE files representing the console keyboard and screen; another predefined file called KEYBOARD also represents the keyboard (see the section below on Predefined Eiles).

UNTYPED FILES

In addition to the standard file types and the INTERACTIVE type, Apple Pascal allows "untyped" files -- objects that are declared with the word FILE and nothing more. Example:

```
VAR F: FILE;
```

Untyped files can only be used with the built-in functions BLOCKREAD and BLOCKWRITE for high-speed data transfers.

An untyped file F can be thought of as a file without a buffer variable F^. All I/O to this file must be accomplished by BLOCKREAD and BLOCKWRITE. These functions are described in the next chapter.

PREDEFINED FILES

The standard predefined files INPUT and OUTPUT refer to the keyboard and the screen respectively. In addition to these, Apple Pascal provides a predefined file called KEYBOARD. The difference between INPUT and KEYBOARD is that when INPUT is used to refer to the keyboard, the typed characters are automatically displayed on the screen; when KEYBOARD is used, the characters are not automatically displayed. This allows a Pascal program to have complete control over the response to characters typed by the user.

All three predefined files are of type INTERACTIVE, and all three are automatically opened via RESET when the Pascal program begins executing.

TEXTFILES

The Apple Pascal system provides that a TEXT or INTERACTIVE diskette file that is created with ".TEXT" as the last part of its title has a special internal format. Such files are called "textfiles" in this manual. Do not confuse textfiles with files that are of type TEXT or INTERACTIVE but do not have titles ending in ".TEXT".

All parts of the Pascal System that deal with files of characters (such as the editor) are designed to use the special textfile format; and if a textfile is accessed by a Pascal program, then the Pascal program will also use the special format. Therefore, the normal procedure is to use a title ending in ".TEXT" whenever you create a diskette file of the Pascal type TEXT or INTERACTIVE. The format of a textfile is as follows:

At the beginning of the file is a 1Ø24-byte header page, which contains information for the use of the text editor. This space is respected by all portions of the system. When a user Pascal program creates a textfile (via REWRITE), the system will automatically create the header. When a user Pascal program accesses an existing textfile (via RESET) the system skips the header. In other words, the header is invisible to a user Pascal program using REWRITE and RESET.

When a program uses BLOCKREAD and BLOCKWRITE to access files, the special textfile structure is <u>not</u> respected.

The system will transfer the header only on a diskette-to-diskette transfer, and will omit it on a transfer to a serial device (thus transfers from diskette to a printer or to the console will omit the header).

Following the header page, the text content itself appears in 1Ø24-byte text pages. Each text page is a sequence of lines, and the last line on a page is followed by enough null characters (ASCII ØØ) to fill out the 1Ø24 bytes. A line is defined as:

```
[DLE indent] [text] CR
```

where the brackets indicate that the DLE and the indent code may be absent and the text itself may be absent.

CR is the "Carriage Return" control character (ASCII 13), and may be absent at the end of the last line in the file. DLE is the "Data Link Escape" control character (ASCII 16). If present it is followed by a code indicating the indentation of the line. The code is 32 + the number of spaces to indent. Thus any leading spaces on a line are replaced by the DLE and the indent code.

The DLE and indent code and the nulls at the end of a text page are, like the header, invisible to a Pascal program. The DLE and indent are automatically translated to leading spaces, and vice versa.

The end of the file is marked by the ETX control character (ASCII 3).

THE SET TYPES

APPLE Pascal supports all of the Standard Pascal constructs for sets. Two limitations are imposed on sets:

- A set may not have more than 512 elements assigned to it.

- A set may not have any INTEGERs less than Ø or greater than 511 assigned to it.

A set of 512 elements will occupy 32 words of memory.

Comparisons and operations on sets are allowed only between sets whose individual elements are of the same type. For example, in the sample program below, the base type of the set S is the subrange type Ø..49, while the base type of the set R is the subrange type 1..1ØØ. The underlying type of both sets is the type INTEGER, so the comparisons and operations on the sets S and R in the following program are legal:

```
PROGRAM SETCOMPARE;
VAR S: SET OF Ø..49;
    R: SET OF 1..1ØØ;

BEGIN
  S:= [Ø,5,1Ø,15,2Ø,25,3Ø,35,4Ø,45];
  R:= [1Ø,2Ø,3Ø,4Ø,5Ø,6Ø,7Ø,8Ø,9Ø];
  IF S = R THEN
    WRITELN('... oops ...')
  ELSE
    WRITELN('sets work');
  S := S + R
END.
```

In the following example, the comparison I = J is not legal since the two sets are of two distinct underlying types.

```
PROGRAM ILLEGALSETS;
TYPE STUFF=(ZERO,ONE,TWO);
VAR I: SET OF STUFF;
    J: SET OF Ø..2;

BEGIN
  I:= [ZERO];
  J:= [1,2];
  IF I = J THEN ...      <<<< error here
END.
```


PACKED VARIABLES

PACK AND UNPACK

Apple Pascal does not require the Standard Pascal procedures PACK and UNPACK, and these procedures are not provided. If a variable is PACKED, all required packing and unpacking are done automatically on an element-by-element basis.

PACKED FILES

Apple Pascal does not support PACKED FILE types. A PACKED FILE can be declared, but the data in the file will not actually be packed.

PACKED ARRAYS

The Apple Pascal compiler supports PACKED ARRAYs as defined in Standard Pascal. For example, consider the following declarations:

```
A: ARRAY[Ø..9] OF CHAR;
B: PACKED ARRAY[Ø..9] OF CHAR;
```

The array A will occupy ten 16-bit words of memory, with each element of the array occupying one word. The PACKED ARRAY B on the other hand will occupy a total of only 5 words, since each 16-bit word contains two 8-bit characters. Each element of B is 8 bits long.

PACKED ARRAYs need not be restricted to arrays of type CHAR. For example:

```
C: PACKED ARRAY[Ø..1] OF Ø..3;
D: PACKED ARRAY[1..9] OF SET OF Ø..15;
D2: PACKED ARRAY[Ø..239,Ø..319] OF BOOLEAN;
```

Each element of the PACKED ARRAY C is only 2 bits long, since only 2 bits are needed to represent the values in the range Ø..3. Therefore C occupies only one 16-bit word of memory, and 12 of the bits in that word are unused. The PACKED ARRAY D is a 9-word array, since each element of D is a SET which can be represented in a minimum of 16 bits. Each element of a PACKED ARRAY OF BOOLEAN, such as D2 in the above example, occupies only one bit.

The details of exactly how variables are packed are unspecified. In most cases, the minimum space into which an array can be packed is one word (two eight-bit bytes). For example, consider

```
BITS: PACKED ARRAY[Ø..7] OF BOOLEAN;
```

This is an eight-element array where each element requires one bit, so you might expect it to occupy eight bits or one byte. In fact, it occupies one word or two bytes. Furthermore, the two-dimensional array

```
BATS: PACKED ARRAY[Ø..3] OF PACKED ARRAY[Ø..7] OF BOOLEAN;
```

or its equivalent

```
BATS: PACKED ARRAY[Ø..3,Ø..7] OF BOOLEAN;
```

consists of four arrays. Each of them, like the previous array, occupies one word. Therefore BATS occupies four words.

Note that a PACKED ARRAY OF CHAR always occupies one byte per character and a PACKED ARRAY OF Ø..255 always occupies one byte per element.

Also, packing never occurs across word boundaries. This means that if the type of element to be packed requires a number of bits which does not divide evenly into 16, there will be some unused bits in each of the words where the array is stored.

The following two declarations are NOT equivalent because of the way the Pascal Compiler is implemented:

```
E: PACKED ARRAY[Ø..9] OF ARRAY[Ø..3] OF CHAR;
F: PACKED ARRAY[Ø..9,Ø..3] OF CHAR;
```

In the declaration of E, the second occurrence of the reserved word ARRAY causes the packing option in the compiler to be turned off. E becomes an unpacked array of 4Ø words. On the other hand, the PACKED ARRAY F occupies only 2Ø words because the reserved word ARRAY occurs only once in the declaration. If E is declared as

```
E: PACKED ARRAY[Ø..9] OF PACKED ARRAY[Ø..3] OF CHAR;
```

or as

```
E: ARRAY[Ø..9] OF PACKED ARRAY[Ø..3] OF CHAR;
```

F and E will have identical configurations.

In declaring a PACKED ARRAY, the word PACKED is only meaningful before the last appearance of the word ARRAY in the declaration. When in doubt, a good rule of thumb for declaring a multidimensional PACKED ARRAY is to place the word PACKED before every appearance of the word ARRAY to ensure that the resultant array will be PACKED.

The array will only be packed if the type of each element of the array is scalar, subrange, or a set and each array element can be represented in 8 bits or fewer. For an array whose elements are sets, this means that the underlying type of the set must not contain more than 8 elements, and must not contain any integer greater than 255.

The following declaration will result in no packing whatsoever because the final type of the array cannot be represented in a field of 8 bits:

```
G: PACKED ARRAY[Ø..3] OF Ø..1ØØØ;
```

G will be an array which occupies four 16-bit words.

Note that a string constant may be assigned to a PACKED ARRAY OF CHAR (if it has exactly the same length), but not to an unpacked ARRAY OF CHAR. Likewise, comparisons between an ARRAY OF CHAR and a string constant are illegal.

Because of their different sizes, PACKED ARRAYs cannot be compared to ordinary unpacked ARRAYs.

A PACKED ARRAY OF CHAR may be printed out with a single write statement (exactly as if it were a string):

```
PROGRAM VERYSLICK;
VAR T: PACKED ARRAY[Ø..1Ø] OF CHAR;
BEGIN
  T:='HELLO THERE';
  WRITELN(T)
END.
```

PACKED RECORDS

The following RECORD declaration declares a RECORD with four fields. The entire RECORD occupies one 16-bit word as a result of declaring it to be a PACKED RECORD.

```
VAR R: PACKED RECORD
         I,J,K: Ø..31;
         B: BOOLEAN
       END;
```

The variables I, J, K each take up five bits in the word. The boolean variable B is allocated to the 16th bit of the same word.

In much the same manner that PACKED ARRAYs can be multidimensional PACKED ARRAYs, PACKED RECORDS may contain fields which themselves are PACKED RECORDS or PACKED ARRAYS. Again, slight differences in the way in which declarations are made will affect the degree of packing

achieved. For example, note that the following two declarations are not equivalent:

```
VAR A:PACKED RECORD                VAR B:PACKED RECORD
      C:INTEGER;                         C:INTEGER;
      F:PACKED RECORD                    F:RECORD
                R: CHAR;                       R:CHAR;
                K: BOOLEAN                     K:BOOLEAN
              END;                           END;
      H:PACKED ARRAY[Ø..3] OF CHAR       H:PACKED ARRAY[Ø..3] OF CHAR
    END;                               END;
```

As with PACKED ARRAYs, the word PACKED should appear with every occurrence of the word RECORD in order for the PACKED RECORD to retain its packed qualities throughout all fields of the RECORD. In the above example, only RECORD A has all of its fields packed into one word. In B, the F field is not packed and therefore occupies two 16-bit words. It is important to note that a packed or unpacked ARRAY or RECORD which is a field of a PACKED RECORD will always start at the beginning of the next word boundary. This means that in the case of A, even though the F field does not completely fill one word, the H field starts at the beginning of the next word boundary.

A case variant may be used as the last field of a PACKED RECORD, and the amount of space allocated to it will be the size of the largest variant among the various cases. The actual nature of the packing is beyond the scope of this document.

```
VAR K: PACKED RECORD
         B: BOOLEAN;
         CASE F: BOOLEAN OF
           TRUE: (Z:INTEGER);
           FALSE: (M: PACKED ARRAY[Ø..3] OF CHAR)
       END;
```

In the above example the B and F fields are stored in two bits of the first 16-bit word of the record. The remaining fourteen bits are not used. The size of the case variant field is always the size of the largest variant, so in the above example, the case variant field will occupy two words. Thus the entire PACKED RECORD will occupy three words.

USING PACKED VARIABLES AS PARAMETERS

No PACKED variable may be passed as a VAR (call-by-reference) parameter to a PROCEDURE or FUNCTION. Packed variables may, however, be passed as ordinary call-by-value parameters.

THE LONG INTEGER TYPE

In Apple Pascal, the predefined INTEGER type can be modified by a length attribute as in the following examples:

```
TYPE BIGNUM = INTEGER[12];
VAR FATS: INTEGER[25];
```

This defines BIGNUM as a type which can have any integer value requiring not more than 12 decimal digits. FATS can have any integer value requiring not more than 25 digits. The length attribute can be any unsigned INTEGER constant up to and including 36.

This is a new kind of type, which is called a LONG INTEGER in this manual. The LONG INTEGER is suitable for business, scientific or other applications which need extended number lengths with complete accuracy. A LONG INTEGER is represented internally as a binary-coded decimal (BCD) number; that is, each decimal digit of the value is represented in binary. This means that there can be no rounding errors in working with LONG INTEGER values.

LONG INTEGER constants are also allowed. Any integer constant whose value exceeds MAXINT is automatically a constant of the type LONG INTEGER.

The integer arithmetic operations (+, -, *, and DIV) can all be used with LONG INTEGER values. However, MOD cannot be used with LONG INTEGERs. In integer arithemetic, overflow occurs if any intermediate or final result requires more than 36 decimal digits. When a LONG INTEGER value is assigned to a LONG INTEGER variable, overflow occurs if the value requires more decimal digits than the defined length of the variable.

An INTEGER value can always be assigned to a LONG INTEGER variable; it is automatically converted to the appropriate length. However, a LONG INTEGER value can never be assigned to an INTEGER variable. If INTEGER and LONG INTEGER values are mixed in an expression, the INTEGER values are converted to LONG INTEGER and the result is a LONG INTEGER value. LONG INTEGERs and REALs are incompatible; they can never be mixed in an arithmetic expression or assigned to each other.

All of the standard relational operators may be used with mixed LONG INTEGER and INTEGER values.

The built-in procedure STR accepts a LONG INTEGER value as a parameter, and converts it to a string of decimal digits. The built-in function TRUNC accepts a LONG INTEGER value as a parameter, and returns the corresponding INTEGER value if the absolute value of the LONG INTEGER is less than or equal to MAXINT. These built-ins are described in the next chapter; they are the only built-ins which accept LONG INTEGER parameters.

An attempt to declare a LONG INTEGER in a parameter list will result in a syntax error. This restriction may be circumvented by defining a type which is a LONG INTEGER. For example:

```
TYPE LONG = INTEGER[18];
PROCEDURE BIGNUMBER(BANKACCT: LONG);
```

EXAMPLES:

```
VAR I: INTEGER;
    L: INTEGER[N]; {where N is an integer constant <= 36 }
    R: REAL;

I:= L {syntax error; the TRUNC function can be used to convert a
       LONG INTEGER to an INTEGER}
L:=-L {correct, if -L does not require more than 36 digits; the
       minus sign doesn't count as a digit}
L:= I {always correct}
L:= R {never accepted}
R:= L {never accepted}
```

The memory space allocated for a LONG INTEGER is always an integral number of words. Specifically, a variable of type INTEGER[n] occupies

```
(n + 3) DIV 4 +1
```

words.

Therefore, the actual limit on the value of a LONG INTEGER may exceed the number of decimal digits specified in its declaration. For example, a length of 5 through 8 occupies three words and can store values up to and including 99999999; a length of 9 through 12 occupies four words and can store values up through 999999999999; a length of 13 through 16 occupies five words and can store values up through 9999999999999999.

CHAPTER 3
BUILT-IN PROCEDURES AND FUNCTIONS

This chapter describes all the built-in procedures and functions of Apple Pascal that differ from Standard Pascal. This does not include the procedures and functions that are provided as library UNITs, e.g. the graphics procedures and functions. Chapter 7 covers the library UNITs provided with Apple Pascal.

Transcendental functions (e.g. the trig functions SIN, COS, etc.) are a special case. In Standard Pascal they are built-in functions, but in Apple Pascal they are in a library UNIT. The ATAN and LOG functions differ slightly from Standard Pascal, and they are described in this chapter. The other transcendentals differ only in that to use them your program must include a USES TRANSCEND statement as described in Chapter 7.

Since some of these built-in procedures and functions do no checking for range validity of parameters, they may easily cause unpredictable results. Those built-ins which are particularily dangerous are noted as such in their descriptions. Any necessary range or validity checks are your responsibility.

STRING BUILT-INS

In the following descriptions, a "string value" means a string variable, a quoted string, or any function or expression whose value is a string. Unless otherwise stated all parameters are called by value.

THE LENGTH FUNCTION

The LENGTH function returns the integer value of the length of a string. The form is

```
LENGTH (STRG)
```

where STRG is a string value. Example:

```
GEESTRING := '1234567';
WRITELN( (LENGTH(GEESTRING), ' ', LENGTH(''))
```

This will print:

```
7  Ø
```

THE POS FUNCTION

The POS function returns an integer value. The form is

```
POS (SUBSTRG, STRG)
```

where both SUBSTRG and STRG are string values. The POS function scans STRG to find the first occurrence of SUBSTRG within STRG. POS returns the index within STRG of the first character in the matched pattern. If the pattern is not found, POS returns zero. Example:

```
STUFF := 'TAKE THE BOTTLE WITH A METAL CAP';
PATTERN := 'TAL';
WRITELN( POS(PATTERN, STUFF) )
```

This will print:

```
26
```

THE CONCAT FUNCTION

The CONCAT function returns a string value. The form is

```
CONCAT ( STRGs )
```

where STRGs means any number of string values separated by commas. This function returns a string which is the concatenation of all the strings passed to it. Example:

```
SHORTSTRING := 'THIS IS A STRING';
LONGSTRING  := 'THIS IS A VERY LONG STRING.';
LONGSTRING := CONCAT('START ', SHORTSTRING, '-', LONGSTRING);
WRITELN(LONGSTRING)
```

This will print:

```
START THIS IS A STRING-THIS IS A VERY LONG STRING.
```

THE COPY FUNCTION

The COPY function returns a string value. The form is

```
COPY (STRG, INDEX, COUNT)
```

where STRG is a string value and both INDEX and COUNT are integer values. This function returns a string containing COUNT characters copied from STRG starting at the INDEXth position in STRG. Example:

```
TL := 'KEEP SOMETHING HERE';
KEPT := COPY(TL, POS('S', TL), 9);
WRITELN(KEPT)
```

This will print:

```
SOMETHING
```

THE DELETE PROCEDURE

The DELETE procedure modifies the value of a string variable. The form is

```
DELETE (STRG, INDEX, COUNT)
```

Where STRG is a string variable called by reference and modified, and both INDEX and COUNT are integer values. This procedure removes COUNT characters from STRG starting at the INDEX specified. Example:

```
OVERSTUFFED := 'THIS STRING HAS FAR TOO MANY CHARACTERS IN IT.';
DELETE(OVERSTUFFED, POS('HAS', OVERSTUFFED)+3, 8);
WRITELN(OVERSTUFFED)
```

This will print:

```
THIS STRING HAS MANY CHARACTERS IN IT.
```

THE INSERT PROCEDURE

The INSERT procedure modifies the value of a string variable. The form is

```
INSERT (SUBSTRG, STRG, INDEX)
```

where SUBSTRG is a string value, STRG is a string variable called by reference, and INDEX is an integer value. This inserts SUBSTRG into STRG at the INDEXth position in STRG. Example:

```
ID := 'INSERTIONS';
MORE := ' DEMONSTRATE';
DELETE(MORE, LENGTH(MORE), 1);
INSERT(MORE, ID, POS('IO', ID));
WRITELN(ID)
```

This will print:

```
INSERT DEMONSTRATIONS
```

THE STR PROCEDURE

The STR procedure modifies the value of a string variable. The form is

```
PROCEDURE STR ( LONG , STRG )
```

where LONG is an integer value, and STRG is a string variable called by reference. LONG may be a LONG INTEGER.

This converts the value of LONG into a string. The resulting string is placed in STRG. See Chapter 2 for more about the use of LONG INTEGERs. Example:

```
INTLONG := 1Ø2Ø395Ø3;
STR(INTLONG, INTSTRING);
INSERT('.', INTSTRING, LENGTH(INTSTRING)-1);
WRITELN('$', INTSTRING)
```

This will print:

```
$1Ø2Ø395.Ø3
```

The following program segment will provide a suitable dollar and cent routine:

```
STR(L,S); INSERT('.',S,LENGTH(S)-1); WRITELN(S)
```

where L and S are appropriately declared.

INPUT AND OUTPUT BUILT-INS

OVERVIEW OF APPLE PASCAL I/O FACILITIES

This section deals with data transfers to and from all peripheral devices, including diskette drives, the screen, the keyboard, printers, etc. There are also certain "integral" devices such as the TTL game-control outputs and the built-in speaker, which are not considered as I/O devices; see Chapter 7. For complete information on Apple Pascal file types, see Chapter 2.

Apple Pascal I/O facilities can be thought of as existing at four different levels:

- Hardware-oriented I/O: the UNITREAD, UNITWRITE, and UNITCLEAR procedures are the lowest level of control. They allow a Pascal program to transfer a specified number of consecutive bytes between memory and a device. They are not controlled by filenames, directories, etc., but merely use device numbers and (for diskette drives) block numbers.

- Untyped file I/O: The BLOCKREAD and BLOCKWRITE functions provide I/O for untyped files (see Chapter 2). They make use of filenames and directories but consider a file to be merely a sequence of bytes -- not a sequence of records of a particular type.

- Typed file I/O: The GET, PUT, and SEEK procedures treat a file as a sequence of records. GET and PUT provide transfers between individual file records and the file's buffer variable, and SEEK moves the pointer to a specified record within the file. The EOF function provides an indication of when the end of the file has been reached.

- Text file I/O: The READ, READLN, WRITE, and WRITELN procedures provide transfers between a file of type TEXT or INTERACTIVE and program variables. The PAGE procedure writes a top-of-form control character into a textfile. The EOLN function provides an indication of when the end of a text line has been reached. This is the highest level of I/O control, with many sophisticated features.

As mentioned in Chapter 2, the INPUT, OUTPUT, and KEYBOARD files are predefined and need not be declared in a program. All other files must first be declared in the VAR section of a program, and must then be opened by means of RESET or REWRITE before they can be used in any way.

Opening a file is a means of associating the file's identifier (declared in the program) with its title (used by the operating system). If the file to be used does not already exist, open it with REWRITE; this causes the operating system to create a directory entry for the file. If REWRITE is used with the title of an existing file, the existing file is destroyed and a new directory entry is created. RESET is used to open an existing file and can also be used to move the file pointer back to the beginning of a file that is already open. A CLOSE procedure is also provided. It offers several options for the disposition of the file when the program is through using it.

If an I/O operation is unsuccessful, the operating system will normally terminate program execution. However, there is a compiler option to disable this feature. The IORESULT function allows the program itself to check on the status of the most recent I/O operation and take appropriate action.

THE REWRITE PROCEDURE

This procedure creates a new file and marks the file as open. As explained below, it can also be used to open an existing file. The form is

```
REWRITE ( FILEID , TITLE )
```

where FILEID is the identifier of a previously declared file, and TITLE is a string containing any legal file title.

If the device specified in the TITLE is not a diskette, then the file is opened for both input and output. If the TITLE indicates a diskette file, REWRITE creates a new file and opens it for input and output.

If the file is already open, an I/O error occurs (see IORESULT below). The file remains open.

An example showing the use of REWRITE in a program follows the description of GET and PUT below.

THE RESET PROCEDURE

This procedure opens an existing file for both reading and writing. There are two forms:

```
RESET ( FILEID , TITLE )
RESET ( FILEID )
```

where FILEID is the identifier of a previously declared file, and TITLE is a string containing any legal file title.

If a TITLE is used and the specified file is already open, an I/O error occurs (see IORESULT, below). The file's state remains unchanged. If the file does not exist, an I/O error occurs.

A RESET without the TITLE can only be used on an open file; the effect is simply to reposition the file pointer as if the file had just been opened.

If the file is not of type INTERACTIVE, RESET automatically performs a GET action -- that is, it loads the first record of the file into the file's buffer variable and advances the file pointer to the second record. If the file is INTERACTIVE, no GET is performed; the buffer variable's value is undefined and the file pointer points to the first record. (GET is described further on.)

Note that RESETting a non-INTERACTIVE file to an output-only device, such as PRINTER:, may cause a run-time error as a result of the automatic GET caused by the RESET.

When an existing file is opened with RESET and is then used for output, only the file records actually written to are affected.

An example showing the use of RESET in a program follows the description of GET and PUT below.

THE CLOSE PROCEDURE

This procedure closes a file which was previously opened with RESET or REWRITE. The form is

```
CLOSE ( FILEID [, OPTION] )
```

where FILEID is the identifier of a previously declared file, and OPTION may be any one of the following:

NORMAL -- a normal close is done, i.e. CLOSE simply sets the file state to closed. If the file was opened using REWRITE and is a disk file, it is deleted from the directory.

LOCK -- the file is made permanent in the directory if the file is on a disk and the file was opened with a REWRITE; otherwise a NORMAL close is done. If the TITLE matches an existing diskette file, the original contents of the file are lost.

PURGE -- if the file is a diskette file, it is deleted from the directory. In the special case of a diskette file that already exists and is opened with REWRITE, the original file remains in the directory, unchanged. If the file is not a diskette file, the associated unit will go off-line.

CRUNCH -- this is like LOCK except that it locks the end-of-file to the point of last access, i.e. everything after the last element accessed is thrown away. If the TITLE matches an existing diskette file, the original contents of the file are lost.

If the OPTION is omitted, the NORMAL close is performed.

All CLOSEs regardless of the option will mark the file closed and will make the file buffer variable FILEID^ undefined. CLOSE on a CLOSEd file causes no action.

An example showing the use of CLOSE in a program follows the description of GET and PUT below.

THE EOF FUNCTION

This function returns a BOOLEAN value to indicate whether the end of a specified file has been reached. When EOF is true, nothing more can be read from the file. The form is

```
EOF [ ( FILEID ) ]
```

If (FILEID) is not present, INPUT is assumed.

EOF is false immediately after the file is opened, and true on a closed file. Whenever EOF (FILEID) is true, FILEID^ is undefined.

After a GET, EOF is true if the GET attempted to access a record that is after the end of the file. After a PUT or WRITE, EOF is true if the file cannot be expanded to accommodate the PUT or WRITE (because of limited diskette space, for example).

For details on EOF after a READ or READLN operation, see the descriptions of READ and READLN further on in this chapter, and Appendix C.

When EOF becomes true during a READ or GET operation, the value of FILFID^ is not defined.

When keyboard input is being read (via the predefined files INPUT or KEYBOARD), EOF only becomes true when the end-of-file character is typed. The end-of-file character is ctrl-C (ASCII 3). EOF remains true until the file INPUT or KEYBOARD is RESET, and no more typed characters can be read until this is done.

An example showing the use of EOF in a program follows the description of GET and PUT below.

THE EOLN FUNCTION

EOLN is defined only for a file of type TEXT, FILE OF CHAR, or INTERACTIVE. This function returns a BOOLEAN value to indicate whether the pointer for a specified text file is at the end of a line. The form is

```
EOLN [ ( FILEID ) ]
```

If (FILEID) is not present, INPUT is assumed.

EOLN returns false immediately after the file is opened, and true on a closed file.

When a GET finds an end-of-line character (the CR character, ASCII 13) in the file, it sets EOLN to true. Instead of loading the end-of-line character into the file's buffer variable it loads a space (ASCII 32).

For the behavior of EOLN after a READ or READLN, see the descriptions of these statements further on.

THE GET AND PUT PROCEDURES

These procedures are used to read or write one logical record from or to a typed file. The forms are

```
GET ( FILEID )
PUT ( FILEID )
```

where FILEID is the identifier of a previously declared typed file. A typed file is any file for which a type is specified in the variable declaration, as opposed to untyped files (see Chapter 2).

GET (FILEID) advances the file pointer to the next record and moves the contents of this record into the file buffer variable FILEID^. The next GET or PUT with the same FILEID will access the next record in sequence.

PUT (FILEID) advances the file pointer to the next record and puts the contents of FILEID^ into this record. The next GET or PUT with the same FILEID will access the next record in sequence.

The actual physical disk access may not occur until the next time the physically associated block of the disk is no longer considered the current working block. The kinds of operation which tend to force the block to be written are: a SEEK to elsewhere in the file, a RESET, and CLOSE. Successive GETs or PUTs to the file will cause the physical I/O to happen when the block boundaries are crossed.

The following two example programs illustrate the use of REWRITE, RESET, CLOSE, EOF, GET, and PUT. The first program creates a new file of type REAL, with the title REAL.DATA, and puts ten REAL values into it. The values are supplied by the user.

To obtain the values, the program uses a WRITE to display a prompt on the screen and a READ to accept the value typed by the user. READ and WRITE are described in detail further on in this chapter.

```
    PROGRAM MAKEFILE;

    VAR F: FILE OF REAL;
        I: INTEGER;

    BEGIN
(*Open with REWRITE since this is a new file.*)
      REWRITE(F, '*REALS.DATA');
(*Read 10 numbers and store them in the file.*)
      FOR I:=1 TO 10 DO BEGIN
(*Put a prompt on the screen.*)
        WRITE('-->');
(*Read a number from the keyboard.*)
        READ(F^);
(*Store the number in the file.*)
        PUT(F)
      END;
(*Close the file and lock it.*)
      CLOSE(F, LOCK)
    END.
```

The second program reads values from the file created by the first program, and displays them on the screen.

```
    PROGRAM READFILE;

    VAR F: FILE OF REAL;

    BEGIN
(*Open with RESET since we want to read the file*)
      RESET(F, '*REALS.DATA');
(*Read each number from the file and display them*)
      WHILE NOT EOF(F) DO BEGIN
(*Display the current number on the screen*)
        WRITELN(F^);
(*Advance to the next number*)
        GET(F)
      END;
(*Close the file*)
      CLOSE(F)
    END.
```

Note that these programs offer no flexibility as to the title of the file. The example under READLN below shows how to let the user specify the title of the file to be used.

THE IORESULT FUNCTION

This function returns an integer value which reflects the status of the last completed I/O operation. The form is

IORESULT

The values returned by IORESULT are as follows (also see Table 2):

Ø	No error; normal I/O completion
1	Bad block on diskette (not used on Apple)
2	Bad device (volume) number
3	Illegal operation (e.g., read from PRINTER:)
4	Unknown hardware error (not used on Apple)
5	Lost device -- no longer on line
6	Lost file -- file is no longer in directory
7	Bad title -- illegal filename
8	No room -- insufficient space on diskette
9	No device -- volume is not on line
1Ø	No such file on specified volume
11	Duplicate file title
12	Attempt to open an already open file
13	Attempt to access a closed file
14	Bad input format -- error in reading real or integer
15	Ring buffer overflow -- input arriving too fast
16	Write-protect error -- diskette is write-protected
64	Device error -- bad address or data on diskette

In normal operation, the Compiler will generate code to perform run-time checks after each I/O operation except UNITREAD, UNITWRITE, BLOCKREAD, or BLOCKWRITE. This causes the program to get a run-time error on a bad I/O operation. Therefore if you want to check IORESULT with your own code in the program, you must disable this compiler feature by using the (*$I-*) option (see Chapter 4).

Note that IORESULT only gives a valid return the first time it is referenced after an I/O operation. If it is referenced again (without another I/O operation), it will always return Ø.

INTRODUCTION TO TEXT I/O

In addition to PUT and GET, Apple Pascal provides the standard procedures READ, READLN, WRITE, and WRITELN, collectively known as the text I/O procedures. They perform the same tasks as in standard Pascal and have the same syntax (with the addition of STRING variables). However, the details of their operation are specific to Apple Pascal and can be complicated. Also, the use of STRING variables and the distinction between TEXT and INTERACTIVE files have important effects.

The text I/O procedures can only be used with files of type TEXT or INTERACTIVE. As already mentioned, RESET makes a distinction between these two file types: when a TEXT file is RESET, a GET is automatically performed but when an INTERACTIVE file is RESET, no GET is performed. This requires READ and READLN to be rather complex procedures. Like many other complex creatures, they will behave simply if you use them simply. Therefore, this manual is written with some assumptions in mind about how they will be used. These assumptions can be translated into the following specific suggestions:

- When using the text I/O procedures don't use GET or PUT, and don't refer explicitly to the file buffer variable F^. The reason is that the text I/O procedures themselves use GET and PUT in complicated ways.

- Don't mix reading and writing operations on the same diskette textfile. If you read from a textfile, CLOSE it and reopen it before writing to it; and vice versa.

- To open an existing diskette textfile for reading, always use RESET. To open an existing diskette textfile for writing, always use REWRITE.

- Don't use READ with a STRING variable. Use READLN.

- Don't use the EOLN function with READLN, and don't use it with STRING variables.

If you follow these suggestions, the text I/O procedures will work exactly as described in the following pages. These are not rules of Pascal; there is nothing in the system that will enforce them. However, the exact details of what happens if you ignore the suggestions are beyond the scope of this chapter.

There may be situations in which these assumptions and suggestions are too restrictive. If so, you will need the complete details on how READ and READLN behave in all possible situations, as given in Appendix C.

In particular, you need the information in Appendix C if you want to mix reading and writing operations or overwrite part of an existing text file without destroying all of the original contents.

THE READ PROCEDURE

This procedure may be used only on TEXT (FILE OF CHAR) or INTERACTIVE files. It allows characters and numeric values to be read from a file without the need for explicit use of GET or explicit reference to the window variable. The form is

PROCEDURE READ ([FILEID,] VBLs)

where FILEID is the identifier of a TEXT or INTERACTIVE file which must

be open. If the FILEID is omitted, INPUT is assumed. VBLs means one or more variables separated by commas. The variables may be of type CHAR, STRING, INTEGER, LONG INTEGER, or REAL. (But you should use READLN for STRING variables).

READ reads values from the file and assigns them to the variables in sequence.

READ With a CHAR Variable

For a CHAR variable, READ reads one character from the file and assigns that character to the variable. There are two special cases: Whenever the end-of-line character (ASCII 13) is READ, the value assigned to the CHAR variable is a space (ASCII 32), not a CR. Whenever EOF becomes true, the value assigned to the CHAR variable is not defined.

After the READ, the next READ or READLN will always start with the character immediately following the one just READ.

The workings of EOLN and EOF depend on whether the file is of type TEXT or INTERACTIVE. For a TEXT file, EOF is true when the last text character in the file has been READ. EOLN is true when the last text character on a line has been READ and whenever EOF is true. (A "text character" here means a character that is not the end-of-line character or the end-of-file character.)

For an INTERACTIVE file, EOF is not true until the end-of-file character has been READ. EOLN is not true until the end-of-line character at the end of the line has been READ or until EOF is true.

If you are using READ with a CHAR variable and you need to use EOLN, you may be able to simplify the situation by using READLN with a STRING variable instead; this gives you line-oriented reading without the need to check EOLN (see below).

READ With a Numeric Variable

For a variable of one of the numeric types, READ expects to read a string of characters which can be interpreted as a numeric value of the same type. Any space or end-of-line characters preceding the numeric string are skipped; and a space, end-of-line, or end-of-file is expected after the numeric string. If a numeric string is not found after skipping spaces and end-of-lines, an I/O error occurs. Otherwise, the string is converted to a numeric value and the value is assigned to the variable.

After the READ, the next READ or READLN will always start with the character immediately following the last character of the numeric string.

If the last character of the numeric string is the last character on the line, then EOLN will be true. If the last character of the numeric string is the last character in the file, then EOF and EOLN will both be true.

If nothing but spaces are found before the EOF, a value of Ø is READ.

Note that the behavior of READ with a numeric variable is exactly the same regardless of whether the file is TEXT or INTERACTIVE.

THE READLN PROCEDURE

This procedure may be used only on TEXT (FILE OF CHAR) or INTERACTIVE files. It allows line-oriented reading of characters, strings, and numeric values. The form is

```
PROCEDURE READLN ( [ FILEID, ] VBLs )
```

where FILEID is the identifier of a TEXT or INTERACTIVE file which must be open. If the FILEID is omitted, INPUT is assumed. VBLs means one or more variables separated by commas. The variables may be of type CHAR, STRING, INTEGER, LONG INTEGER, or REAL.

READLN works exactly like READ, except that after a value has been read for the last variable, the remainder of the line is skipped (including the end-of-line). After any READLN, the next READ or READLN will always start with the first character of the next line, if there is a next line. If there is no next line, EOF will be true.

READLN with a STRING variable reads all the characters up to but not including the end-of-line character. Thus repeated READLN's with a STRING variable have the effect of reading successive lines of the file as strings.

One of the most common uses of READLN with a STRING variable is to read a string of characters from the CONSOLE: device. In the following example, which is a modification of the previous example under GET and PUT, READLN is used to read a filename typed by the user:

```
PROGRAM MAKEFILE;

VAR F: FILE OF REAL;
    I: INTEGER;
    TITLE: STRING;

BEGIN
(*Ask user for title.*)
   WRITE('Type name of file: ');
(*Accept line typed by user.*)
   READLN(TITLE);
(*If title has no suffix, add .DATA suffix.*)
   IF POS('.', TITLE)=Ø THEN TITLE:=CONCAT(TITLE, '.DATA');
(*Open with REWRITE since this is a new file*)
   REWRITE(F, TITLE);
   ...
(*Remainder of program is identical to previous example.*)
```

Another useful example is given below under WRITE and WRITELN.

THE WRITE AND WRITELN PROCEDURES

These procedures may be used only on TEXT (FILE OF CHAR) or INTERACTIVE files. They allow characters, strings, and numeric values to be written to a file without the need for explicit use of PUT or explicit reference to the window variable. Also, WRITELN allows line-oriented output. The forms are

```
WRITE ( [ FILEID, ] [ ITEMs ] )
WRITELN [ ( [ FILEID, ] [ ITEMs ] ) ]
```

where FILEID is the identifier of a TEXT or INTERACTIVE file which must be open. If the FILEID is omitted, OUTPUT is assumed.

ITEMs means one or more ITEMs separated by commas. Each ITEM has one of the following forms:

```
EXPR

    or

EXPR : FIELDWIDTH

    or

EXPR : FIELDWIDTH : FRACTIONLENGTH
```

where EXPR is an expression whose value is to be written, FIELDWIDTH is an INTEGER expression which specifies the minimum number of characters to be written, and FRACTIONLENGTH is an INTEGER expression which specifies the number of digits to be written after the decimal point if EXPR is of type REAL. The default FRACTIONLENGTH is 5; the default FIELDWIDTH is 1. For a non-negative REAL value, one space is always written before the first digit; for a negative REAL value, the minus sign occupies this position.

WRITE evaluates the expressions and writes their values to the file in sequence. If EXPR is of type CHAR, STRING, or PACKED ARRAY of CHAR, WRITE writes the character(s) to the file and advances the file pointer. If a FIELDWIDTH has been given and the number of characters written is less than specified, leading spaces are added to fill the field.

If EXPR is of a numeric type, WRITE converts the value to a string of characters in standard Pascal numeric format, writes this string to the file, and advances the pointer. If the value is REAL and a FRACTIONLENGTH has been given, the specified number of digits are written after the decimal point; if no FRACTIONLENGTH is given, five decimal places are written. If necessary, the value is rounded (not truncated) to the number of decimal places available. If a FIELDWIDTH has been given and the number of characters written is less than specified, leading spaces are added to fill the field.

WRITELN works exactly like WRITE, except that after the last value has been written a return character is written to end the line. This allows line-oriented output with string expressions.

OUTPUT is the identifier of a predeclared INTERACTIVE file which can be used with WRITE and WRITELN. All characters written to OUTPUT are displayed on the console screen. When a program is writing to OUTPUT, the user may type ctrl-S to stop the output. The program halts until another character is typed, then resumes the output where it left off. Also, the user may type ctrl-F. This halts the displaying of characters on the console screen, but the program continues to run.

The following example program illustrates a number of useful techniques. It uses line-oriented I/O with STRING variables, but performs character manipulations on the STRING variables. It also shows a useful trick for opening a file for output which may or may not exist already. The effect of the program is to read the input file line by line, remove any leading periods from the lines, and write the lines out to the output file.

```
PROGRAM FLUSHPERIODS;

CONST PERIOD='.';

VAR INFILE, OUTFILE: TEXT;
    INNAME, OUTNAME, LINEBUF: STRING;
```

```
    BEGIN

(*First get the files open.*)
(*Get input filename.*)
       WRITE('Name of input file: ');
       READLN(INNAME);
(*Supply the default suffix .TEXT if needed.*)
       IF POS('.', INNAME)=Ø THEN INNAME:=CONCAT(INNAME,'.TEXT');

(*Turn off automatic error checking so program can do it.*)
       (*$I-*)
(*Input file should already exist, so open with reset.*)
       RESET(INFILE,INNAME);
(*If it doesn't work, complain and stop program.*)
       IF IORESULT<>Ø THEN BEGIN
         WRITELN('File not found.');
         EXIT(PROGRAM)
       END;
(*Turn automatic error checking back on.*)
       (*$I+*)

(*Get output filename.*)
       WRITE('Name of output file: ');
       READLN(OUTNAME);
(*Supply default suffix .TEXT if needed.*)
       IF POS('.', OUTNAME)=Ø THEN OUTNAME:=CONCAT(OUTNAME, '.TEXT');
(*Open file with rewrite.*)
       REWRITE(OUTFILE,OUTNAME);

(*Now do the job.*)
       WHILE (NOT EOF(INFILE)) AND (NOT EOF(OUTFILE)) DO BEGIN
         READLN(INFILE,LINEBUF);
         IF LENGTH(LINEBUF) >Ø THEN
           IF POS(PERIOD, LINEBUF)=1 THEN DELETE(LINEBUF, 1, 1);
         WRITELN(OUTFILE,LINEBUF)
       END;

(*Now clean up.*)
(*If the output file isn't complete...*)
       IF EOF(OUTFILE) THEN BEGIN
         WRITELN('Not enough room in output file!');
(*...Then throw it away.*)
         CLOSE(OUTFILE,PURGE)
       END
(*If it's okay, then lock it into the directory.*)
       ELSE CLOSE(OUTFILE,LOCK);
       CLOSE(INFILE)
    END.
```

THE PAGE PROCEDURE

This procedure sends a top-of-form character (ASCII 12) to the file. The form is

```
PAGE ( FILEID )
```

where FILEID is the identifier of an open file of type TEXT or INTERACTIVE.

THE SEEK PROCEDURE

This procedure allows the program to move a file pointer to any specified record in a file that is not a textfile. This allows random access to file records. The form is

```
SEEK ( FILEID , RECNUM )
```

where FILEID is the identifier of an open file that is not a textfile(i.e. not created with the .TEXT suffix), and RECNUM is an integer value interpreted as a record number in the file.

This procedure changes the file pointers so that the next GET or PUT from/to the file uses the record of FILEID specified by RECNUM. Records in files are numbered from Ø. A GET or PUT must be executed between SEEK calls since two SEEKs in a row may cause unexpected, unpredictable junk to be held in the window and associated buffers. Immediately after a SEEK, EOF will return false; a following GET or PUT will cause EOF to return the appropriate value.

The following sample program demonstrates the use of SEEK to randomly access and update records in a file:

```
    PROGRAM RANDOMACCESS;
(*Allows update of any selected record in a file.*)
    VAR
      RECNUMBER: INTEGER;
      FNAME: STRING;
      VITALS: FILE OF RECORD
                        NAME: STRING[2Ø];
                        DAY,MONTH,YEAR: INTEGER;
                        ADDRESS: STRING[5Ø];
                        ALIVE: BOOLEAN
                      END;
```

```
    BEGIN
(*Obtain filename.*)
      WRITE('Enter filename: ');
      READLN(FNAME);
(*Use RESET to preserve existing contents of file; but if it doesn't
exist, use REWRITE to create it.*)
      (*$I-*)
      RESET(VITALS, FNAME);
      IF IORESULT<>Ø THEN REWRITE(VITALS, FNAME);
      (*$I+*)

(*Repeat the following "forever," i.e. until EXIT is caused by user
typing ctrl-C and causing EOF(INPUT), or by lack of diskette space for
new records.*)
      WHILE TRUE DO BEGIN
(*Obtain record number; quit if user types ctrl-C, causing EOF.*)
        WRITE('Enter record number: ');
        READLN(RECNUMBER);
        IF EOF THEN BEGIN
                      CLOSE(VITALS, LOCK);
                      EXIT(PROGRAM)
                    END;

(*GET the specified record*)
        SEEK(VITALS,RECNUMBER);
        GET(VITALS);

(*Update the record*)
        WITH VITALS^ DO BEGIN
          WRITELN(NAME);
          WRITE('Enter correct name: ');
          READLN(NAME);
          WRITELN(DAY);
          WRITE('Enter correct day: ');
          READLN(DAY);
(*...and so forth with other fields of record.*)
        END;

(*Now SEEK the same record again, since the GET advanced the file
pointer to the next record after it got the current record into
VITALS^ *)
        SEEK(VITALS,RECNUMBER);

(*PUT updated record into file; exit if this causes EOF.*)
        PUT(VITALS);
        IF EOF(VITALS) THEN BEGIN
                              WRITELN('Not enough file space!');
                              EXIT(PROGRAM)
                            END
      END
    END.
```


THE UNITREAD AND UNITWRITE PROCEDURES

THESE ARE DANGEROUS PROCEDURES

These are the low-level procedures which do device-oriented I/O. The forms are

```
UNITREAD ( UNITNUMBER, ARRAY, LENGTH [, [BLOCKNUMBER] [, MODE]] )
UNITWRITE ( UNITNUMBER, ARRAY, LENGTH [, [BLOCKNUMBER] [, [MODE]] )
```

where:

UNITNUMBER, an integer, is the volume number of an I/O device. The Apple Pascal Operating System Reference Manual describes these numbers.

ARRAY is the name of a packed array, which may be subscripted to indicate a starting position. This is used as the starting address to do the transfers from/to. A string may also be used, but it should have a subscript greater than Ø, since the Øth element of a string contains data which usually should not be transmitted.

LENGTH is an integer value designating the number of bytes to transfer.

BLOCKNUMBER, an integer, is meaningful only when using a disk drive and is the absolute block number at which the transfer will start. If the BLOCKNUMBER is left out, Ø is assumed.

MODE, an integer in the range Ø..15, is optional; the default is Ø. It controls two UNITWRITE options which are described below. MODE has no effect on UNITREAD.

The UNITWRITE options controlled by the MODE parameter apply only to text-oriented I/O devices such as the console or a printer; they do not apply to diskette drives. Both options are enabled by default, if no MODE parameter is supplied.

One option is conversion of DLE codes. In a Pascal textfile, any leading spaces at the beginning of a line are represented by a DLE character (ASCII 16) followed by a code value which is 32 plus the number of spaces. On output to a non-block-structured device such as a printer, the DLE conversion option detects the DLE code and converts it into a sequence of spaces.

Conversion of DLE codes is disabled by a MODE value that has a one in Bit 3 (see below).

The other option is automatic line feeds. In a Pascal textfile, the end of each line is marked by the end-of-line character CR (ASCII 13) without any line-feed character. On output to a non-block-structured

device such as a printer, the automatic line-feed option inserts an LF character (ASCII 1Ø) after every CR character (ASCII 13).

Automatic line feeds are disabled by a MODE value that has a one in Bit 2 (see below).

Only Bit 2 and Bit 3 of the MODE value have any significance. Bit 2, by itself, corresponds to a value of 4, and Bit 3 by itself corresponds to a value of 8. The following values can be used to control the options:

- MODE=Ø (the default value) causes both options to be enabled.
- MODE=4 causes automatic line feeds to be disabled, while leaving DLE conversion enabled.
- MODE=8 causes DLE conversion to be disabled, while leaving automatic line feeds enabled.
- MODE=12 disables both DLE conversion and automatic line feeds.

THE UNITBUSY FUNCTION

This is a UCSD Pascal procedure used to indicate whether a specified device is busy. But since the I/O drivers on the Apple are not interrupt driven, UNITBUSY will always return the value FALSE. To test whether a character is available from the Apple keyboard, use the KEYPRESS function (see Chapter 7).

THE UNITWAIT PROCEDURE

This is a UCSD Pascal procedure which waits for a specified device to complete the I/O in progress. But since the I/O drivers on the Apple are not interrupt driven, UNITWAIT does nothing.

THE UNITCLEAR PROCEDURE

This procedure cancels all I/O operations to the specified unit and resets the hardware to its power-up state. The form is

```
UNITCLEAR ( UNITNUMBER )
```

IORESULT is set to a non-zero value if the specified unit is not present (you can use this to test whether or not a given device is present in the system). The form

```
UNITCLEAR (1)
```

flushes the type-ahead buffer for CONSOLE: and resets horizontal scrolling to full left (displays leftmost 4Ø characters on Apple's screen).

THE BLOCKREAD AND BLOCKWRITE FUNCTIONS

These functions transfer data to or from an untyped file. They return an integer value which is the number of blocks of data actually transferred. The forms are

```
BLOCKREAD  ( FILEID, ARRAYNAME, BLOCKS [, RELBLOCK] )
BLOCKWRITE ( FILEID, ARRAYNAME, BLOCKS [, RELBLOCK] )
```

where

FILEID must be the identifier of a previously declared untyped file.

ARRAYNAME is the identifier of a previously declared array. The length of the array should be an integer multiple of 512. ARRAYNAME may be indexed to indicate a starting position in the array.

BLOCKS is the number of blocks to be transferred.

RELBLOCK is the block number relative to the start of the file, the zero-th block being the first block in the file. If no RELBLOCK is specified, the reads/writes will be done sequentially. Specifying RELBLOCK moves the file pointer.

WARNING: Caution should be exercised when using these functions, as the array bounds are not heeded. EOF(FILEID) becomes true when the last block in a file is read.

The following program illustrates the use of BLOCKREAD and BLOCKWRITE.

```
PROGRAM FILEDEMO;

VAR
  BLOCKNUMBER,BLOCKSTRANSFERRED:INTEGER;
  BADIO: BOOLEAN;
  G,F: FILE;
  BUFFER:  PACKED ARRAY[Ø..511] OF CHAR;

 (* This program reads a diskfile called 'SOURCE.DATA' and copies
 the file into another diskfile called 'DESTINATION' using untyped
 files and the built-ins BLOCKREAD and BLOCKWRITE *)

BEGIN
  BADIO:=FALSE;
  RESET(G,'SOURCE.DATA');
  REWRITE(F,'DESTINATION');
  BLOCKNUMBER:=Ø;
  BLOCKSTRANSFERRED:=BLOCKREAD(G,BUFFER,1,BLOCKNUMBER);
  WHILE (NOT EOF(G)) AND (IORESULT=Ø) AND (NOT BADIO) AND
        (BLOCKSTRANSFERRED=1) DO
    BEGIN
      BLOCKSTRANSFERRED:=BLOCKWRITE(F,BUFFER,1,BLOCKNUMBER);
      BADIO:=((BLOCKSTRANSFERRED<1) OR (IORESULT<>Ø));
      BLOCKNUMBER:=BLOCKNUMBER+1;
      BLOCKSTRANSFERRED:=BLOCKREAD(G,BUFFER,1,BLOCKNUMBER)
    END;
  CLOSE(F,LOCK)
END.
```


MISCELLANEOUS BUILT-INS

THE ATAN FUNCTION

The ATAN function is simply a different identifier for the ARCTAN function of Standard Pascal. Along with the other transcendental functions, it is part of the TRANSCEND UNIT supplied with Apple Pascal (see Chapter 7).

THE LOG FUNCTION

This function returns a real value which is the logarithm (base 1Ø) of its argument. Along with the other transcendental functions, it is part of the TRANSCEND UNIT supplied with Apple Pascal (see Chapter 7). The form is

```
LOG ( NUMBER )
```

where NUMBER can be either a real or an integer value.

THE TRUNC FUNCTION

The function TRUNC will accept a LONG INTEGER as well as a REAL as an argument. Overflow will result if the absolute value of the argument exceeds MAXINT. With a REAL argument, TRUNC returns an INTEGER value formed by dropping the fractional part of the REAL value. With a LONG INTEGER value, TRUNC returns a numerically equivalent INTEGER value.

THE PWROFTEN FUNCTION

This function returns a real value which is 1Ø to a specified (integer) power. The form is

```
PWROFTEN ( EXPONENT )
```

where EXPONENT is an integer value in the range Ø..37. This function returns the value of 1Ø to the EXPONENT power.

THE MARK AND RELEASE PROCEDURES

The Standard Pascal procedure DISPOSE is not provided in Apple Pascal. Instead, the MARK and RELEASE procedures are used for returning dynamic memory allocations to the system. The forms are

```
MARK ( HEAPPTR )
RELEASE ( HEAPPTR )
```

where HEAPPTR is of type ^INTEGER and is called by reference in the MARK procedure. MARK sets HEAPPTR to the value of the system's current top-of-heap pointer. RELEASE sets the system's top-of-heap pointer to the value of HEAPPTR.

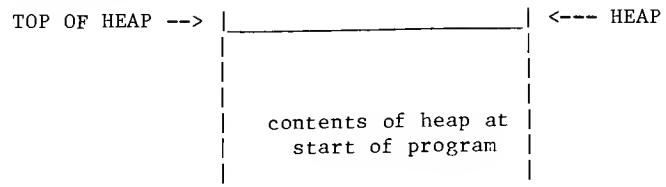
The process of recovering memory space described below is only an approximation to the function of DISPOSE as one cannot explicitly ask that the storage occupied by one particular variable be released by the system for other uses.

Variables created by the standard procedure NEW are stored in a stack-like structure called the "heap". The following program is a simple demonstration of how MARK and RELEASE can be used to change the size of the heap.

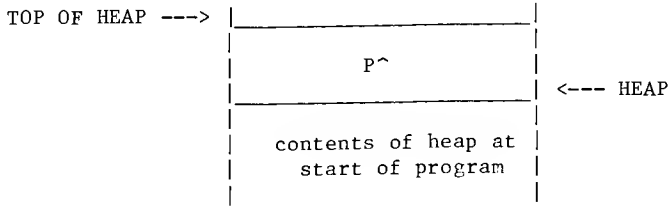
```
PROGRAM SMALLHEAP;

TYPE  PERSON=
        RECORD
          NAME: PACKED ARRAY[Ø..15] OF CHAR;
          ID: INTEGER
        END;

VAR P: ^PERSON;
    HEAP: ^INTEGER;

BEGIN
  MARK(HEAP);
  NEW(P);
  P^.NAME:='FARKLE, HENRY J.';
  P^.ID:= 999;
  RELEASE(HEAP)
END.
```

The program shows a particularly handy method for deliberately accessing the contents of memory which is otherwise inaccessable. It first calls MARK to place the address of the current top of heap into the variable HEAP.

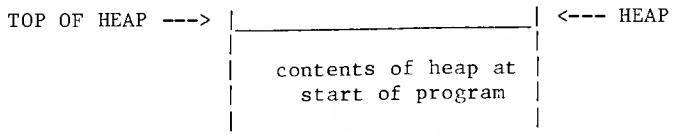
Below is a pictorial description of the heap at this point in the program's execution:



Next the program calls the standard procedure NEW and this results in a new variable P^ which is located in the heap as shown in the diagram below:



After the RELEASE the heap is as follows:



Once the program no longer needs the variable P^ and wishes to "release" this memory space to the system for other uses, it calls RELEASE which resets the top of heap to the address contained in the variable HEAP.

If NEW had been called several times between the calls to MARK and RELEASE, the storage occupied by several variables would have been RELEASEd at once. Note that because of the stack nature of the heap it is not possible to release the memory space used by a single item in the middle of the heap.

Careless use of MARK and RELEASE can leave "dangling pointers", pointing to areas of memory which are no longer part of the defined heap space.

THE HALT PROCEDURE

This procedure generates a HALT opcode that, when executed, causes a non-fatal run-time error to occur. The form is

```
HALT
```

For a more orderly way to terminate program execution, see EXIT below.

THE EXIT PROCEDURE

The EXIT procedure causes an orderly exit from a procedure or function, or from the program itself. The forms are

```
EXIT(procedurename)
EXIT(programname)
EXIT(PROGRAM)
```

In the first form shown, EXIT accepts as its single parameter the identifier of a procedure or function to be exited. Note that this need not be the procedure or function in which the EXIT statement occurs. EXIT follows the trail of procedure or function calls back to the procedure or function specified; each procedure or function in the trail is exited. If the specified procedure is recursive, the most recent invocation of that procedure will be exited.

When a procedure or function is exited via EXIT, any files local to it are automatically closed, just as if it had terminated normally.

The use of EXIT to exit a function can cause the function to return an undefined value if no assignment to the function identifier is made before EXIT is executed.

When the program name or the reserved word PROGRAM is used as the parameter for EXIT, EXIT brings the program to an orderly halt.

THE MEMAVAIL FUNCTION

This function returns the number of words currently between the top-of-stack and top-of-heap. This can be interpreted as the amount of memory available at that time. The form is

```
MEMAVAIL
```


THE GOTOXY PROCEDURE

This procedure sends the cursor to a specified position on the screen. The form is

```
GOTOXY ( XCOORD , YCOORD )
```

where XCOORD and YCOORD are integer values interpreted as X (horizontal) and Y (vertical) coordinates. XCOORD must be in the range Ø through 79; YCOORD must be in the range Ø through 23. The cursor is sent to these coordinates. The upper left corner of the screen is assumed to be (Ø,Ø).

This procedure is written to work with the Apple II's screen. If you wish to use an external terminal, you will need to bind in a new GOTOXY using the BINDER package described in the Pascal Operating System Manual.

THE TREESEARCH FUNCTION

This a fast function for searching a binary tree that has a particular kind of structure. The form is

```
TREESEARCH(ROOTPTR, NODEPTR, NAME)
```

where ROOTPTR is a pointer to the root node of the tree to be searched, NODEPTR is a pointer variable to be updated by TREESEARCH, and NAME is the identifier of a PACKED ARRAY[1..8] OF CHAR which contains the 8-character name to be searched for in the tree.

The nodes of the binary tree are assumed to be linked records of the form

```
NODE=RECORD
       NAME: PACKED ARRAY[1..8] OF CHAR;
       LEFTLINK, RIGHTLINK: ^NODE;

           ...(*other fields can be anything*)...

     END;
```

The type name and the field names are not important; TREESEARCH only assumes that the first eight bytes of the record contain an 8-character name and are followed by two pointers to other nodes.

It is also assumed that names are not duplicated within the tree and are assigned to nodes according to an alphabetical rule: for a given node, the name of the left subnode is alphabetically less than the name of the node, and the name of the right subnode is alphabetically greater than the name of the node. Finally, any links that do not point to other nodes should be NIL.

TREESEARCH can return any of three values:

Ø: The NAME passed to TREESEARCH has been found in the tree. NODEPTR now points to the node with the specified name.

1: The NAME is not in the tree. If it is added to the tree, it should be the right subnode of the node pointed to by NODEPTR.

-1: The NAME is not in the tree. If it is added to the tree, it should be the left subnode of the node pointed to by NODEPTR.

The TREESEARCH function does not perform any type checking on the parameters passed to it.

BYTE-ORIENTED BUILT-INS

These procedures and functions are all byte-oriented. The system does not protect itself from them, as no range checking of any sort is performed on the parameters and no type checking is performed on the source and destination parameters. Read the descriptions carefully before trying them out. Also, some machine dependencies may lurk in the implementations.

THE SIZEOF FUNCTION

This function returns an integer value, which is the number of bytes occupied by a specified variable, or by any variable of a specified type. SIZEOF is particularly useful for FILLCHAR, MOVERIGHT, and MOVELEFT built-ins (see below). The form is

```
SIZEOF ( IDENTIFIER )
```

where IDENTIFIER is either a type identifier or a variable identifier.

THE SCAN FUNCTION

This function scans a range of memory bytes, looking for a one-character target. The target can be a specified character, or it can be any character that does not match the specified character. SCAN returns an integer value, which is the number of bytes scanned. The form is

```
SCAN ( LIMIT , PEXPR , SOURCE )
```

where

LIMIT is an integer value which gives the maximum number of bytes to scan. If LIMIT is negative, SCAN will scan backward. If SCAN fails to find the specified target, it will return the value of LIMIT.

PEXPR is a "partial expression" which specifies the target of the scan. PEXPR takes one of the following forms:

```
= CH          (target is a character equal to CH)
<> CH         (target is a character not equal to CH)
```

where CH stands for any expression that yields a result of type char.

SOURCE is a variable of any type except a file type. The first byte of the variable is the starting point of the scan.

SCAN terminates when it finds the target or when it has scanned LIMIT bytes. It then returns the number of bytes scanned. If the target is found at the starting point, the value returned will be zero. If LIMIT is negative, the scan will go backward and the value returned will also be negative.

Examples: Suppose that DEM is declared as follows:

```
VAR DEM: PACKED ARRAY [Ø..1ØØ] OF CHAR;
```

and then the first 53 elements of DEM are loaded with the characters

```
.....THE PTERO IS A MEMBER OF THE PTERODACTYL FAMILY.
```

We then have the following:

```
SCAN(-26,=':',DEM[3Ø])      will return -26

SCAN(1ØØ,<>'.',DEM)         will return 5

SCAN(15,=' ',DEM[5])        will return 3.
```

THE MOVELEFT AND MOVERIGHT PROCEDURES

These procedures do mass moves of a specified number of bytes. The forms are

```
MOVELEFT ( SOURCE , DESTINATION , COUNT )
MOVERICHT ( SOURCE , DESTINATION , COUNT )
```

where SOURCE and DESTINATION are two variables of any type except a file type. The first byte of SOURCE is the beginning of the range of bytes whose values are copied. The first byte of DESTINATION is the beginning of the range of bytes that the values are copied into. COUNT is an integer and specifies the number of bytes moved.

MOVELEFT starts from the left end of the SOURCE range. It proceeds from left to right, copying bytes into DESTINATION, starting at the left end of the DESTINATION range.

MOVERIGHT starts from the right end of the SOURCE range. It proceeds from right to left, copying bytes into DESTINATION, starting at the right end of the DESTINATION range.

The reason for having both of these is that the SOURCE and DESTINATION ranges may overlap. If they overlap, the order in which bytes are moved is critical: each byte must be moved before it gets overwritten by another byte.

In general this consideration applies when SOURCE and DESTINATION are subarrays of the same PACKED ARRAY OF CHAR. If bytes are being moved to the right (DESTINATION has a higher subscript than SOURCE), use MOVERIGHT. If bytes are being moved to the left (DESTINATION has a lower subscript than SOURCE), use MOVELEFT.

THE FILLCHAR PROCEDURE

This procedure fills a specified range of memory bytes with a specified character value. The form is

```
FILLCHAR ( DESTINATION , COUNT , CHARACTER )
```

where DESTINATION is a variable of any type except a file type. The first byte of DESTINATION is the beginning of the range of bytes to be filled. COUNT is an integer value and specifies the number of bytes to be filled. CHARACTER is a character value to be copied into each byte in the specified range.

SUMMARY

STRING BUILT-INS

Integer-Valued Functions:

LENGTH (STRG) returns length of string.
POS (SUBSTRG , STRG) returns index of first occurence of SUBSTRG within STRG.

String-Valued Functions:

CONCAT (STRGs) returns concatenation of strings.
COPY (STRG , INDEX , COUNT) returns a substring of STRG.

Procedures:

DELETE (STRG , INDEX , COUNT) deletes a substring of STRG.
INSERT (SUBSTRG , STRG , INDEX) inserts a substring into STRG.
STR (LONG , STRG) converts integer or long integer to string of decimal digits.

INPUT AND OUTPUT BUILT-INS

Opening and Closing Files:

RESET (FILEID [, TITLE]) opens existing diskette file, or resets pointers to beginning if already open.
REWRITE (FILEID , TITLE) opens new diskette file.
CLOSE (FILEID [, OPTION]) closes file. OPTION may be LOCK, NORMAL, PURGE, or CRUNCH. Default is NORMAL.

File Pointer Status:

EOF [(FILEID)] boolean, true if end of file has been reached or file is closed. Default FILEID is INPUT.
EOLN [(FILEID)] boolean, true if end of line has been reached. Default FILEID is INPUT.
SEEK (FILEID , INTEGER) moves file pointer to specified record.

Typed File I/O:

GET (FILEID) reads current file record into window & advances file pointer. Default FILEID is INPUT.
PUT (FILEID) writes window into current file record & advances file pointer. Default FILEID is OUTPUT.
IORESULT returns an integer value which depends on status of most recent I/O operation. Value is zero for OK completion.
READ ([FILEID,] VBLs) where VBLs means one or more variables separated by commas. Successive values are read from file into variables. Default FILEID is INPUT. FILEID must be of type TEXT (FILE OF CHAR) or INTERACTIVE.
READLN ([FILEID,] VBLs) Like READ, but skips to beginning of next line after reading value for last VBL.
WRITE ([FILEID,] [EXPRs]) where EXPRs means one or more expressions separated by commas. Each EXPR may also be followed by field width and number of decimal places. Expression values are written to successive file records. Default FILEID is OUTPUT. FILEID must be of type TEXT (FILE OF CHAR) or INTERACTIVE.
WRITELN [([FILEID,] [EXPRs])] Like WRITE, but writes an end-of-line after last EXPR value.
PAGE (FILEID) writes a top-of-form (ASCII 12).

Device I/O:
These built-ins are described in detail in the text.

```
UNITREAD ( UNITNUMBER , ARRAY , LENGTH [, [BLOCKNUMBER] [, MODE]] )
UNITWRITE ( UNITNUMBER , ARRAY , LENGTH [, [BLOCKNUMBER] [, MODE]] )
UNITBUSY ( UNITNUMBER ) : BOOLEAN
UNITWAIT ( UNITNUMBER )
UNITCLEAR ( UNITNUMBER )
```

Untyped File I/O:
These built-ins are described in detail in the text.

```
BLOCKREAD (FILEID, ARRAY, BLOCKS [, RELBLOCK]) : INTEGER
BLOCKWRITE (FILEID, ARRAY, BLOCKS [, RELBLOCK]): INTEGER
```

MISCELLANEOUS BUILT-INS

ATAN (NUMBER) returns a REAL value. This is the ARCTAN function of Standard Pascal. NUMBER may be REAL or INTEGER.
LOG (NUMBER) returns a REAL value, the log base 10 of NUMBER. NUMBER may be REAL or INTEGER.
TRUNC (NUMBER) returns an INTEGER value. This is like Standard Pascal except that NUMBER may be LONG INTEGER instead of REAL. NUMBER may not exceed MAXINT.
PWROFTEN (EXPONENT) returns a REAL value which is 10 to the EXPONENT power. EXPONENT is an INTEGER in the range 0..37.
MARK (HEAPPTR) where HEAPPTR is of type ^INTEGER. HEAPPTR is called by name and is set to current top-of-heap.
RELEASE (HEAPPTR) where HEAPPTR is of type ^INTEGER. The current top-of-heap pointer is set to HEAPPTR.
HALT causes non-fatal run-time error; halts program.
EXIT causes orderly exit from procedure, function, or program.
MEMAVAIL returns an INTEGER value, the number of words between top-of-stack and top-of-heap.
GOTOXY (XCOORD , YCOORD) moves screen cursor to specified coordinates. XCOORD is an INTEGER in the range 0..79 and YCOORD is an INTEGER in the range 0..23.
TREESEARCH (ROOTPTR , NODEPTR , NAME) searches for NAME in a binary tree. See text for details.

BYTE-ORIENTED BUILT-INS

These built-ins are described in detail in the text.

SIZEOF (VARIABLE OR TYPE IDENTIFIER)
SCAN (LIMIT , PEXPR , SOURCE)
MOVELEFT (SOURCE , DESTINATION , COUNT)
MOVERIGHT (SOURCE , DESTINATION , COUNT)
FILLCHAR (DESTINATION , COUNT , CHARACTER)

CHAPTER 4
THE PASCAL COMPILER

INTRODUCTION

The purpose of the Apple Pascal Compiler is to translate the text of a Pascal program into the compressed P-code version of the program. This P-code is the "machine language" of the UCSD Pascal interpreter or "pseudo-machine," described in the Apple Pascal Operating System Manual.

Complete details on operation of the Compiler are in the Pascal Operating System Reference Manual; the following two sections on diskette files needed and on using the Compiler are somewhat abridged.

DISKETTE FILES NEEDED

To operate the Pascal Compiler, you need the following diskette files:

Textfile to be Compiled	(Any diskette, any drive; default is boot diskette's text workfile SYSTEM.WRK.TEXT, any drive)
SYSTEM.COMPILER	(Any diskette, any drive)
SYSTEM.LIBRARY	(Boot diskette, any drive; required only if any of the UNITs in the system library are USEd by the program. See Chapter 5.)
Other Libraries	(Any diskette, any drive; required if any UNITs not in the system library are USEd by the program being compiled. See Chapter 5.)
SYSTEM.EDITOR	(Any diskette, any drive; optional; to fix errors found by Compiler)
SYSTEM.SYNTAX	(Boot diskette, any drive; optional messages given on entering Editor)

In addition to the above files, the following files may be needed if you are invoking the Compiler automatically via the R(un command (see Apple Pascal Operating System Reference Manual for details):

SYSTEM.LINKER
SYSTEM.PASCAL
SYSTEM.CHARSET

One-drive note: The files SYSTEM.COMPILER, SYSTEM.EDITOR, and SYSTEM.SYNTAX are all on diskette APPLEØ:, which is the normal one-drive boot diskette. If you have been working on a program in the Editor, and U(pdating the workfile, your boot diskette has all the files needed to R(un or C(ompile the workfile. If you wish to R(un or C(ompile a textfile that is not already on the boot diskette, use the Filer's T(ransfer command to transfer that textfile onto your boot diskette before compiling. If your program requires Linking to external routines, see the Apple Pascal Operating System Manual.

Multi-drive note: The files SYSTEM.EDITOR and SYSTEM.SYNTAX are both on diskette APPLE1:, which is the normal multi-drive boot diskette. The file SYSTEM.COMPILER is on diskette APPLE2:, which is normally kept in drive volume #5: in a multi-drive system. With APPLE1: in the boot drive and APPLE2: in a non-boot drive, your system has all the files needed to R(un or C(ompile the workfile.

Two-drive note: If you wish to R(un or C(ompile a textfile that is not already on APPLE1: or APPLE2:, and your system has only two drives, use the Filer's T(ransfer command to transfer that textfile onto either APPLE1: or APPLE2: before compiling. Another possibility for two-drive systems is to make APPLEØ: your boot diskette (just put APPLEØ: in the boot drive and press the Apple's RESET key). This frees your second drive to hold a source or destination diskette for compilations, saving you from T(ransferring the source file onto APPLE1: or APPLE2:. APPLEØ: does not contain SYSTEM.LINKER; if your program requires Linking to external routines, use APPLE1: and APPLE2:.

USING THE COMPILER

The Compiler is invoked by typing C for C(ompile or R for R(un from the outermost Command level of the Pascal system. The screen immediately shows the message

```
COMPILING...
```

The Compiler automatically compiles the .TEXT part of the workfile and saves the resulting code (if compilation is successful) as the .CODE part of the workfile. If there is a workfile, but you do not wish to compile that file, use the Filer's N(ew command to clear away the workfile before compiling. If no workfile is available, you are prompted for a source filename:

```
COMPILE WHAT TEXT?
```

You should respond by typing the name of the text file that you wish to have compiled. Do NOT type the suffix .TEXT -- that suffix is automatically supplied by the Compiler, in addition to any suffix you may specify.

Next, if there is no workfile, you will be asked for the name of the file where you wish to save the compiled version of your program:

```
TO WHAT CODEFILE?
```

If you simply press the RETURN key the command will not be terminated, as you might expect. Instead, the source file will be compiled and the compiled version of your program will be saved on the boot diskette's

workfile SYSTEM.WRK.CODE. This is handy if you then wish to R(un the program.

If you want the compiled version of your program to have the same name as the text version of your program (of course, the suffix will be .CODE instead of .TEXT), just type a dollar sign and press the RETURN key. This is a handy feature, since you will usually want to remember only one name for both versions of your program. The dollar sign repeats your entire source file specification, including the volume identifier, so do NOT specify the volume identifier before typing the dollar sign. Note that this use is different from the use of the dollar sign in the Filer.

If you want your program stored under another filename, type the desired filename. Do NOT type the suffix .CODE -- that suffix is automatically supplied by the Compiler, in addition to any suffix you may specify.

By default, the compiler places the code file at the beginning of the largest unused space on the diskette. To override this, you can give a size specification with the filename. In this case, you DO type the suffix .CODE, followed by the number of blocks in square brackets, followed by a period:

```
TO WHAT CODEFILE? MYPROG.CODE[8].
```

The period at the end prevents the system from adding the .CODE prefix after the size specification. The size specification [8] causes the code file to be placed in the first location on the diskette where at least 8 blocks are available.

While the compiler is running, messages on the screen show the progress of the compilation as in the following example:

```
PASCAL COMPILER II.1 [B2B]
<   Ø>......
TUNAFISH  [  2334 WORDS]
<   6>........
14 LINES
SMALLEST AVAILABLE SPACE = 2334 WORDS
```

The identifiers appearing on the screen are the identifiers of the program and its procedures. The identifier for a procedure is displayed at the moment when compilation of the procedure body is started.

The numbers within [ ] indicate the number of (16-bit) words available for symbol table storage at that point in the compilation. If this number ever falls below 55Ø, the compiler will fail. You must then put the swapping option (described below) into your program and recompile.

The numbers enclosed within < > are the current line numbers. Each dot on the screen represents one source line compiled.

If the Compiler detects an error in your program, the screen will show the text preceding the error, an error number, and a marker <<<< pointing to the symbol in the source where the error was detected. The following is an example:

```
[ <<<<
LINE 9, ERROR 18: <SP>(CONTINUE), <ESC>(TERMINATE), E(DIT
```

This shows that the bracket [is an illegal symbol at this point in the program. You have three options when you see a message like this. Pressing the spacebar instructs the Compiler to continue the compilation, in case you want to find more of the errors right now. Pressing the ESC key causes termination of the compilation and return to the Command level.

Typing E sends you to the Editor, which automatically reads in the workfile, ready for editing. If you were not compiling the workfile, the Editor requests the name of the file you were compiling. You should respond by typing the filename of the file you were compiling, and that file will then be read into the Editor. When the correct file has been read into the Editor, the top line of the screen displays the error message (or number, if SYSTEM.SYNTAX was not available) and the cursor is placed at the symbol where the error was detected.

If SYSTEM.SYNTAX is not available, you can look up the error in Table 6 of Appendix B. (You may wish to delete the file SYSTEM.SYNTAX to obtain more diskette space.)

THE COMPILER OPTIONS

COMPILER OPTION SYNTAX

Compiler options (see the following section for details) are placed in the text to be compiled, and take effect when the Compiler arrives at the option during compilation.

Compiler options look like a special kind of comment, and take the following form:

```
(*$option*)
```

The Compiler treats material between (*$ and *) as a compiler option. As shown below, there must be no spaces in (*$ or immediately after the $ character:

```
(*$G-*)          This is a compiler option.
(* $G-*)         This is a comment.
(*$ G-*)         This is a comment.
```

Several options can be combined in one set of (*$...*) brackets, by separating the options with commas (don't add extra spaces):

(*$option,option*) Example: (*$I-,S+,G-*)

You can't do this with the options that involve names or strings of characters.

A given option may be turned on or off at any point in the compilation. The compilation is affected only from the point where the option is turned on until the point where the option is turned off again. Thus you can turn an option on (or off) just during the compilation of a particular routine in your program.

Some options require a filename immediately following the option letter, instead of the usual + or -. In this case, all characters between the option letter and the closing *) are taken as the filename, except that blanks preceding or following the filename are ignored. Therefore, the filename must be the last item before the *). If the first character of a filename is + or -, you must place a blank between the option letter and the filename. For examples of specifying a filename, see the section describing the Include-file mechanism.

In Apple Pascal, curly brackets { and } are equivalent to the normal comment or option delimiters (* and *). The curly brackets cannot be generated by the Apple keyboard, so no confusion exists for programs written on the Apple computer. However, other terminals may be able to generate the curly brackets in programs. These programs will be executed correctly on the Apple, but the curly brackets will be displayed on Apple's screen as square brackets [and] .

THE "COMMENT" OPTION

This option consists of the letter C and a line of text. The text is placed, character for character, in Block Ø of the codefile (where it will not affect program operation). The purpose of this is to allow a copyright notice or other comment to be embedded in the codefile. Example:

(*$C COPYRIGHT ALLUVIAL O. FANSOME 1979*)

The Comment option must precede the heading statement of the program.

THE "GOTO STATEMENTS" OPTION

Tells the Compiler whether to allow or forbid the use of the Pascal GOTO statement within a program.

Default value: G-

(*$G+*) Allows the use of the GOTO statement.

(*$G-*) Causes the Compiler to treat a GOTO as an error.

Teachers sometimes use the G- option to keep novice programmers from using the GOTO statement where more structured approaches using FOR, WHILE, or REPEAT statements would be more appropriate.

THE "IO CHECK" OPTION

This option tells the compiler whether or not to create error-checking code after each structured file I/O statement (not the BLOCK or UNIT I/O statements).

Default value: I+

(*$I+*) Instructs the Compiler to generate code after each statement which performs any I/O, in order to check that the I/O operation was accomplished successfully. In the case of an unsuccessful I/O operation, the program will be terminated with a run-time error.

(*$I-*) Instructs the Compiler not to generate any I/O-checking code. In the case of an unsuccessful I/O operation, the program is not terminated with a run-time error.

The (*$I-*) option is useful for programs where I/O checking is not desirable, or which do their own checking via the IORESULT function. The program can then detect and report the I/O errors, without being terminated abnormally with a run-time error. However, the disadvantage of setting the (*$I-*) option is that I/O errors, (and possibly severe program bugs), may go undetected.

THE "INCLUDE FILE" OPTION

The syntax for instructing the Compiler to include another source file into the compilation is as follows:

(*$I filename *)

All characters between (*$I and *) are taken as the filename of the source file to be included. Thus, the filename must be the last item

before the *). Spaces preceding the filename and following it are ignored.

Note that if the first character of a filename is + or -, you MUST place a blank space between (*$I and the filename. Also, you may not use the * or *: notation in the filename to specify the boot diskette.

If the initial attempt to open the file which is being included (also called the "include file") fails, the Compiler concatenates the suffix .TEXT to the filename and tries again. If this second attempt fails, or if some I/O error occurs while reading the include file, the Compiler responds with a fatal error message and terminates its operation.

If the include file option occurs within the body of a procedure or within the body of the main program, the includ file must not contain any USES, LABEL, CONST, TYPE, or VAR declarations. Otherwise, the compiler accepts include files which contain such declarations even though the declarations of the original program have already been compiled.

The Compiler cannot keep track of nested include options, i.e. an include file must not contain an include file option. This results in a fatal Compiler error.

The include file option makes it easier to compile large programs without having the entire source in one very large file. This is especially useful when the combined file would be too large to edit in the existing Editor's buffer.

THE "LISTING" OPTION

Controls whether the Compiler will generate a program listing.

Default value: L-

(*$L+*) Instructs the Compiler to save a compiled listing on the boot diskette under the filename SYSTEM.LST.TEXT.

(*$L-*) Tells Compiler to make no compiled listing.

(*$L filename*) Tells Compiler to save compiled listing in the specified file.

For example, the following will cause the compiled listing to be sent to the printer:

```
(*$L PRINTER:*)
```


The following will cause the compiled listing to be sent to a diskfile called DEMO1.TEXT on the diskette named MYDISK:

```
(*$L MYDISK:DEMO1.TEXT *)
```

The specified filename, which must be the last item before the *), is used exactly as typed. No suffix is added. Note that a diskette listing file may be edited just like any other text file, provided the filename which is specified contains the suffix .TEXT.

In the compiled listing, the Compiler places next to each source line the line number, segment number, procedure number, and the number of bytes or words (bytes for code, words for data) required by that procedure's declarations or code to that point. The Compiler also indicates whether the line lies within the actual code to be executed or is a part of the declarations for that procedure by printing a "D" for declaration and an integer Ø..9 to designate the level of statement nesting within the code part.

Here is a sample listing, to which column headings have been added:

```
Source  Segment  Procedure  Lexical  Byte number          Program
 line    number   number  : level   within procedure     text

   1       1        1:D       1     (*$L SCRATCH:LIST1.TEXT*)
   2       1        1:D       1
   3       1        1:D       1     PROGRAM DOCTOR;
   4       1        1:D       3     VAR DAY,CURE:INTEGER;
   5       1        1:D       5
   6       1        2:D       1     PROCEDURE DOSE;
   7       1        2:Ø       Ø     BEGIN
   8       1        2:1       Ø       WRITE('AN APPLE A DAY');
   9       1        2:1      26       WRITE(' AND ')
  1Ø       1        2:Ø      43     END;
  11       1        2:Ø      56
  12       1        3:D       1     PROCEDURE TREATMENT;
  13       1        3:Ø       Ø     BEGIN
  14       1        3:1       Ø       FOR CURE:=1 TO 4 DO
  15       1        3:2      11       BEGIN
  16       1        3:3      11         DOSE
  17       1        3:2      11       END
  18       1        3:Ø      13     END;
  19       1        3:Ø      34
  2Ø       1        1:Ø       Ø     BEGIN
  21       1        1:1       Ø       FOR DAY:=Ø TO 25 DO
  22       1        1:2      13       BEGIN
  23       1        1:3      13         TREATMENT;
  24       1        1:3      15         WRITELN('')
  25       1        1:2      35       END
  26       1        1:Ø      35     END.
```

The information given in the compiled listing can be very valuable for debugging a large program. A run-time error message will indicate the segment number, procedure number, and the offset (byte number within the

current procedure) where the error occurred.

Here is a sample run-time error message:

```
EXEC ERR # 1Ø
S# 1, P# 7, I# 56
TYPE <SPACE> TO CONTINUE
```

where S# is the segment number, P# is the procedure number, and I# is the byte number in that procedure where the error occurred.

THE "NOLOAD" OPTION

This option prevents the code of a UNIT used by the program (see Chapter 5) from being kept in memory when the program is executed. Instead, the UNIT's code is in memory only when some portion of it is active.

Default value: N-

(*$N+*) UNIT code will be loaded only when active.

(*$N-*) UNIT code will be loaded as soon as program begins executing.

The (*$N*) option should be placed at the beginning of the main program. Note that use of the (*$N+*) option does not prevent the initialization portion of a UNIT from being executed.

THE "PAGE" OPTION

If a listing is being produced, the P option causes one form-feed (ASCII 12) to be inserted into the text of the listing, just before the line containing the P option. Eor example, if your program contains the line

```
(*$P*)
```

that line will appear at the top of a new page when you print the program's compiled listing.

THE "QUIET COMPILE" OPTION

The Q Compiler option is the "quiet compile" option which can be used to suppress the screen messages that tell the procedure names and line numbers and detail the progress of the compilation.

Default value: Q-

(*$Q+*) Causes the Compiler to suppress output to the screen.

(*$Q-*) Causes the Compiler to send procedure name and line number messages to the screen.

This is mostly useful when the CONSOLE: device is not the Apple's TV or monitor screen, for example if you are using a printing terminal. In normal use with your Apple, this option is not needed.

THE "RANGE CHECK" OPTION

With the (*$R+*) option, the Compiler will produce code which checks on array and string subscripts and on assignments to variables of subrange and string types.

Default value: R+

(*$R+*) Turns range checking on.

(*$R-*) Turns range checking off.

Note that programs compiled with the (*$R-*) option selected will run slightly faster. However if an invalid index occurs or an invalid assignment is made, the program will not be terminated with a run-time error. Since you should never assume that a program is absolutely correct, use (*$R-*) only when speed is critical.

THE "RESIDENT" OPTION

This option forces the code of a specified UNIT or SEGMENT procedure to be kept in memory, for as long as the procedure that contains the option is active. It can thus override the automatic swapping out of a SEGMENT PROCEDURE or EUNCTION (see Chapter 5), and the automatic swapping out of a UNIT caused by the NOLOAD option (see above). Eor example, suppose that MOBY is a large SEGMENT PROCEDURE. Normally it is in memory only when it is active (thus allowing the memory space to be used for something else). But another procedure, RATS, calls MOBY repeatedly. We don't want the disk drive to be whizzing MOBY in and out of memory each time RATS calls it, so we make MOBY a "resident procedure" within RATS:

```
PROCEDURE RATS (HATS, BATS, CATS:INTEGER);
  VAR EOON, MOON: STRING;
  BEGIN
    (*$R MOBY*)
    .
    .
    .
```

Now MOBY will be kept in memory as long as RATS is active. The resident option must immediately follow the BEGIN that starts the procedure body.

The resident option is also useful in connection with the noload option described above.

THE "SWAPPING" OPTION

This option determines whether or not the Compiler operates in "swapping" mode. There are two main parts of the Compiler: one processes declarations; the other handles statements. In the S+ swapping mode, only one of these parts is in main memory at a time. This makes about 3900 additional words available for symbol-table storage at the cost of slower compilation speed (approximately 300 lines/minute in S- mode, versus about 150 lines/minute in S+ mode) because of the overhead of swapping the Compiler segments in from disk. This option must occur before the Compiler encounters any Pascal syntax.

Default value: S-

(*$S+*) Puts Compiler in swapping mode.

(*$S-*) Puts Compiler in non-swapping mode.

(*$S++*) Compiler does even more swapping than with the S+ option. The program compiles still more slowly, but still more room is left in memory for symbol-table storage.

The S+ option should be used when compiling a UNIT.

THE "USER PROGRAM" OPTION

This option determines whether this compilation is a user program compilation, or a compilation of a system program.

Default value: U+

(*$U+*) Informs the Compiler that this compilation is to take place on the user program lex level.

(*$U-*) Tells the Compiler to compile the program at the system lex level. This setting of the U compiler option also causes the following options to be set: R-, G+, I-

NOTE: The U- option will generate programs that do not behave as expected. Not recommended for non-systems work unless you know its method of operation.

THE "USE LIBRARY" OPTION

This option consists of the letter U and a filename. The named file becomes the library file in which subsequent USEed UNITs are sought. The specified filename, which must be the last item before the *), is used exactly as typed. No suffix is added.

The default filename for the library is SYSTEM.LIBRARY, on the boot diskette. If any USEd UNITs are in the boot diskette's SYSTEM.LIBRARY, and you refer to those UNITs first, you do not need the Use-library Compiler option for those UNITs. See Chapter 5 for more details on UNITs.

Following is an example of a valid USES clause employing the U filename Compiler option:

```
USES UNIT1,UNIT2, (*FOUND IN *SYSTEM.LIBRARY*)
(*$U MYDISK:A.CODE *) UNIT3,
(*$U APPLE1:B.LIBRARY *)
UNIT4,UNIT5;
```

Note: In a U filename option, you may not use the * or *: notation to specify the boot diskette.

Some programs require the Compiler to access another diskette file -- for example, an "include" file. When this is done, 2K of memory is required for the diskette directory. If the program is very large, this additional memory is not available and the compilation fails. If this happens to you, try the following technique:

Use the Filer command M(ake to create a 4-block file named SYSTEM.SWAPDISK on the same diskette that contains the Compiler. Now, when the Compiler reads a diskette file during compilation, it will write out 2K of information from memory to SYSTEM.SWAPDISK, thus freeing 2K of memory for the diskette directory. When the diskette directory is no longer needed, the 2K of information is read back into memory from SYSTEM.SWAPDISK.

COMPILER OPTION SUMMARY

All Compiler options are placed in the source text in "dollar-sign comments":

```
(*$option*)                Examples: (*$G-*)
                                     (*$I TURTLE.TEXT *)
```

Compiler-option specifications may be combined in one set of (*$...*) brackets:

```
(*$option,option*)         Example: (*$F-,S+,G+*)
```

If a filename is specified, it must be the last item before the *).

```
C             Following characters are placed directly into codefile.

G+            Allows GOTO statements.
G-            Forbids GOTO statements (default).

I+            Generates I/O-checking code (default).
I-            No I/O checking.
I filename    Includes named source file in compilation.

L+            Sends compiled listing to SYSTEM.LST.TEXT, on boot disk.
L-            Makes no compiled listing (default).
L filename    Sends compiled listing to named file.

N+            Prevents UNITs from being loaded until activated.
N-            Loads UNITs immediately when program runs (default).

P             Inserts a page-feed into compiled listing.

Q+            Suppresses screen messages.
Q-            Sends procedure names and line numbers to CONSOLE: (default)

R+            Generates range-checking code (default).
R-            No range checking.
R name        Keeps named procedure loaded while current one is active.

S+            Puts Compiler in swapping mode.
S++           Compiler does even more swapping.
S- Non-swapping mode.

U+            Compiles user program (default).
U-            Compiles system program.
U filename    Specifies name of library file for finding UNITs.
```


CHAPTER 5
PROGRAMS IN PIECES

INTRODUCTION

Apple Pascal supports the separation of procedures and functions, or groups of them, from the main program. When you are developing a large or complex program, this can be very useful as it allows you to reduce the size of code files, to reduce the memory space used by the program, and to use a set of procedures and functions in more than one program.

Separation can be achieved both at the P-code level and at the source-language level. At the P-code level, any procedure or function can be designated as a SEGMENT. The result is that its code is not loaded into memory until it is called by some other part of the program. As soon as the SEGMENT procedure or function is no longer active it is "swapped out;" that is, its memory space is made available for some other use such as dynamic memory allocation or swapping in another SEGMENT. This technique is sometimes called "overlaying."

At the source-language level, a group of one or more procedures or functions can be compiled separately as a UNIT. The result of compiling a UNIT is a library file; it can either be used directly or incorporated into some other library file such as SYSTEM.LIBRARY.

Separate compilation has several advantages in the development of any large or complicated program, because it allows you to approach the task as a group of smaller tasks which are linked together in a simple and logical way. Several of the powerful features of Apple Pascal are implemented as UNITs, as we will see in Chapter 7. To use a separately compiled UNIT, a program must contain a USES declaration with the name of the UNIT; the program is then called a host program.

There are two kinds of UNITs: Regular UNITs and Intrinsic UNITs. When a host program USES a Regular UNIT, the UNIT's code is inserted into the host program's codefile by the Linker. This need only be done once unless the UNIT is modified and recompiled; then it must be relinked into the host program.

When a host program USES an Intrinsic UNIT, the UNIT's code remains in its library file and is automatically loaded into memory when the host program is executed. This keeps the size of the host program's codefile down, which is particularly important if many programs use the UNIT. It also allows the UNIT to be modified and recompiled without the need to relink.

The Compiler's NOLOAD and RESIDENT options (see Chapter 4) allow further control over the handling of Intrinsic UNITs and SEGMENT procedures and functions. NOLOAD prevents any UNIT from being automatically loaded until its code is activated by the host program. The RESIDENT option can modify the effect of NOLOAD or of a SEGMENT procedure or function; it forces a procedure or function to be kept in memory over a specified range of program execution -- specifically, as long as the procedure or function containing the RESIDENT option is active, the procedure named in the RESIDENT option is kept in memory.

Finally, there is the EXTERNAL mechanism. This allows a procedure or function to be declared in a Pascal host program, without any statements except a heading and the word EXTERNAL. The procedure or function is implemented separately in assembly language, assembled, and then linked into the host program with the Linker. This can be advantageous for procedures or functions which must run very fast.

SEGMENT PROCEDURES AND FUNCTIONS

Declarations of SEGMENT procedures and functions are identical to ordinary Pascal procedures and functions except that the word PROCEDURE or FUNCTION is preceded by the word SEGMENT. For example:

```
SEGMENT PROCEDURE INITIALIZE;
BEGIN
  (* Pascal statements *)
END;

SEGMENT FUNCTION FFT(DOMAIN:MPTR): NPTR;
BEGIN
  (* Pascal statements *)
END;
```

Program behavior does not differ; however, the code and data for a SECMENT procedure or function are in memory only while the procedure or function is actually running. This can be modified by use of the Compiler option (*$R name*) as explained in Chapter 4.

Any procedure or function definition may have the word SECMENT. This includes FORWARD definitions and nested definitions.

The advantage of using SECMENT procedures is the ability to fit large programs into the available memory. To write such a program, divide it into two or more main tasks which are implemented as SECMENT procedures. To be effective, each SEGMENT should be substantial in size and the program should be designed so that SEGMENTs are not swapped in and out too frequently.

REQUIREMENTS AND LIMITATIONS

The disk which holds the code file for the program must be on line (and in the same drive as when the program was started) whenever one of the SEGMENT procedures is to be called. Otherwise, the system will attempt to retrieve and execute whatever information now occupies that particular location on the disk now in that drive, usually with very displeasing results.

SEGMENT procedures must be the first procedure declarations that contain code-generating statements.

LIBRARIES AND UNITS

So far, we have seen Pascal programs which are compiled into codefiles; a codefile can be R(un or eX(ecuted. Now we will consider UNITs, which are compiled into libraries. Two or more libraries can be combined into one file. A library is not R(un or eX(ecuted; instead, it is used by one or more programs.

A library contains code for procedures and/or functions which are available to any program that uses the library, just as if they were defined in the program itself. For example, the Apple Pascal System comes with a library called SYSTEM.LIBRARY which contains code for several UNITs; one of the UNITs is called TURTLEGRAPHICS, and it provides a set of procedures and functions for high-resolution graphics on the Apple. To use these procedures and functions, a program need only have the line

```
USES TURTLEGRAPHICS;
```

after the program heading. The program can then use a TURTLEGRAPHICS procedure such as TURNTO or MOVE.

You can create and compile your own UNITs, and either add them to SYSTEM.LIBRARY or build your own libraries by using the LIBRARY utility described in the Apple Pascal Operating System Reference Manual.

If a UNIT used by your program is contained in the SYSTEM.LIBRARY file, a R(un command will automatically invoke the Linker to do the necessary linking. Otherwise, you must explicitly invoke the Linker. Note that if the UNIT is not contained in the SYSTEM.LIBRARY file, you must use the (*$U filename*) option of the compiler to tell the compiler which library file contains the unit. The (*$U filename*) is placed anywhere before the appearance of the UNIT name in the USES declaration.

UNITS AND USES

The source text for a UNIT has a form somewhat similar to a Pascal program, as explained in detail further on. Briefly, it consists of four parts:

- A heading.
- An INTERFACE part which defines the way the host program communicates with the procedures and functions of the UNIT.
- An IMPLEMENTATION part which defines the procedures and functions themselves.
- An "initialization" which consists of a BEGIN and an END with any number of statements between them. This is the "main program" of the UNIT, and is automatically executed at the beginning of the host program. Note that the initialization

may consist of just the BEGIN and END, with no statements between them.

There are two different flavors of UNITs called Regular and Intrinsic.

Regular UNITs

The heading of a Regular UNIT has the form

```
UNIT name;
```

The UNIT is linked into the host program just once after the program is compiled, and the entire UNIT's code is actually inserted in the host program's codefile at that time.

Intrinsic UNITs

Intrinsic UNITs can only be used by installing them in the SYSTEM.LIBRARY file. This is done after compilation by using the LIBRARY utility program (see Apple Pascal Operating System Reference Manual).

An Intrinsic UNIT is "pre-linked," and its code is never actually inserted into the host program's codefile. When you R(un the host program, the Linker is not called and does not have to be on line. The Intrinsic UNIT's code is loaded into memory when the host program is to be executed. Thus an intrinsic UNIT can be used in many different programs, but there is only one stored copy of the UNIT's code.

This can be especially useful when writing for a one-drive system which does not have room for the Linker or for large programs on the main system diskette. Note that the SYSTEM.LIBRARY file must be on line each time the calling program is executed.

The heading of an Intrinsic UNIT has the form

```
UNIT name;
INTRINSIC CODE csegnum [DATA dsegnum];
```

where csegnum and dsegnum are the segment numbers to be associated with the UNIT in when it is installed in the SYSTEM.LIBRARY file. You choose these numbers, and the system uses them to identify the UNIT at run time. Segment numbers range from Ø to 31, but certain numbers between Ø and 15 must not be used (see below). The UNIT will generate a data segment if it declares any variables not contained in procedures or functions.

The code segment will be associated with segment csegnum and its data segment (if there is one) will be associated with segment dsegnum.

Every unit in a library has a specific segment number associated with it. The segment numbers used by items already in the library are shown in parentheses by the LIBRARY and LIBMAP utility programs (see Apple Pascal Operating System Reference Manual). In choosing segment numbers for an Intrinsic UNIT, the constraint is that when the host program runs, the segment numbers used by the program must not conflict. Observe the following:

- While any program is executing, the system uses segment Ø and the main program body uses segment 1. Therefore, never use these numbers for an Intrinsic UNIT.
- Segments 2 through 6 are reserved for use by the system.
- If the program declares any SEGMENT procedures or functions, these procedures or functions use sequentially increasing segment numbers starting at 7.
- Each UNIT used by the program uses the segment number shown in the library listing.
- If possible, avoid any duplication of segment numbers in the library.

Generally, it is a good idea to use segment numbers in the range from 16 through 31.

The compiler's SWAPPING option,

```
(*$S+*)
```

should always be used when a UNIT is compiled. It should precede the heading of the UNIT.

The INTERFACE Part of a UNIT

The first part of a UNIT is the INTERFACE.

The INTERFACE part immediately follows the UNIT's heading line. It declares constants, types, variables, procedures and functions that are public -- that is, the host program can access them just as if they had been declared in the host program. The INTERFACE portion is the only part of the UNIT that is "visible" from the outside; it specifies how a host program can communicate with the UNIT.

Procedures and functions declared in the INTERFACE are abbreviated to nothing but the procedure or function name and the parameter specifications, as shown in the example below.

The IMPLEMENTATION Part of a UNIT

The IMPLEMENTATION part immediately follows the last declaration in the INTERFACE.

The IMPLEMENTATION begins by declaring those labels, constants, types, variables, procedures and functions that are private -- that is, not accessible to the host program. Then the public procedures and functions that were declared in the INTERFACE are defined. As shown in the example below, they are defined without parameters or function result types, since these have already been defined in the INTERFACE.

The Initialization Part of a UNIT

At the end of the IMPLEMENTATION part, following the last function or procedure, there is the "initialization" part. This is a sequence of statements preceded by BEGIN and terminated with END. The resulting code runs automatically when the host program is executed, before the host program is run. It can be used to make any preparations that may be needed before the procedures and functions of the UNIT can be used. For example, the initialization part of the TRANSCEND UNIT in SYSTEM.LIBRARY generates a table of trigonometric values to be used by the transcendental functions. If you don't want any initialization to take place, you must still have the END followed by a period.

AN EXAMPLE UNIT

Let's sketch out an imaginary Intrinsic UNIT that needs a DATA segment, to demonstrate the information given above.

```
(*$S+*)              (* Swapping is required for compiling UNITs *)

UNIT FROG; INTRINSIC CODE 25 DATA 26;

    INTERFACE                                  (* This stuff is public *)
        CONST FLYSIZE = 10;
        TYPE WARTTYPE = (GREEN,BROWN);
        VAR FROGNAME:STRING[20];          (* Will need Data segment *)
        PROCEDURE JUMP(DIST:INTEGER);
        FUNCTION WARTS:INTEGER;


    IMPLEMENTATION                             (* This stuff is private *)
        CONST PI = 3.14159;
        TYPE ETC = 0..13;
        VAR FROGLOC:INTEGER;

        PROCEDURE JUMP;                  (* Note: no parameters here *)
          BEGIN
            FROGLOC := FROGLOC + DIST
          END;

        FUNCTION WARTS;
          BEGIN
            (* Function definition here *)
          END;

        (* More procedures and functions here *)


    BEGIN
        (* Initialization code, if any, goes here *)
    END.
```

Variables of type FILE must be declared in the INTERFACE part of a UNIT. A FILE declared in the IMPLEMENTATION part will cause a syntax error upon compilation.

USING THE EXAMPLE UNIT

The UNIT above, properly completed, would then be compiled. Then the UNIT would be installed in SYSTEM.LIBRARY, using the LIBRARY utility. Once in the library, the UNIT could then be used by any Pascal host program. A sample program to use our UNIT is sketched out below:

```
PROGRAM JUMPER;

    USES FROG;
    CONST ... ;
    TYPE ... ;
    VAR ... ;
    PROCEDURE ... ;
    FUNCTION ... ;

    BEGIN
      ... ;
      ... ;
      ... ;
    END.
```

A program must indicate the UNITs that it USES before the declaration part of the program; procedures and functions may not contain their own USE declarations. At the occurrence of a USES declaration, the Compiler references the INTERFACE part of the UNIT as though it were part of the host text itself. Therefore all constants, types, variables, functions, and procedures publicly defined in the UNIT are global. Name conflicts may arise if the user defines an identifier that has already been publicly declared by the UNIT. If the UNIT is not in the SYSTEM.LIBRARY, the USES declaration must be preceded by a "use library" option to tell the compiler what library file contains the UNIT.

NESTING UNITS

A UNIT may also USE another UNIT, in which case the USES declaration must appear at the beginning of the INTERFACE part. For example, our UNIT FROG might use the graphics UNIT TURTLEGRAPHICS:

```
(*$S+*)
UNIT FROG; INTRINSIC CODE 25 DATA 26;

    INTERFACE
        USES TURTLEGRAPHICS;
        CONST FLYSIZE = 1Ø;
          . . . . .
             etc.
```

When you later use such a UNIT, your host program must declare that it USES the deepest nested UNIT first:

```
PROGRAM JUMPER;

    USES TURTLEGRAPHICS,FROG;
```

There is one limitation: an Intrinsic UNIT cannot USE a Regular UNIT.

CHANGING A UNIT OR ITS HOST PROGRAM

For test purposes, you may define a Regular UNIT right in the host program, after the heading of the host program. In this case, you will compile both the UNIT and the host program together. Any subsequent changes in the UNIT or host program require that you recompile both.

Normally, you will define and compile a Regular UNIT separately and use it as a library file (or store it in another library by using the LIBRARY utility). After compiling a host program that uses such a UNIT, you must link that UNIT into the host program's codefile by executing the Linker. Trying to R(un an unlinked code file will cause the Linker to run automatically, looking for the UNIT in the system library. Trying to X(ecute an unlinked file causes the system to remind you to link the file.

Changes in the host program require that you recompile the host program. You must also link in the UNIT again, if it is not Intrinsic.

Changes in a Regular UNIT require you to recompile the UNIT, and then to recompile and relink all host programs (or other UNITs) which use that UNIT.

INTRINSIC UNITs and their host programs can be changed as described above, but they do not have to be relinked.

EXTERNAL PROCEDURES AND FUNCTIONS

EXTERNAL procedures (.PROC's) are separately assembled assembly-language procedures, often stored in a library file. Host programs that require EXTERNAL procedures must have them linked into the compiled code file.

A host program declares that a procedure (or function) is EXTERNAL in much the same way as a procedure is declared FORWARD. A standard heading is provided, followed by the keyword EXTERNAL:

```
PROCEDURE FRAMMIS (WIDGET, GIDIBRION:INTEGER);
EXTERNAL;
```

There is one special rule for the heading of an EXTERNAL procedure or function: A VAR parameter can be declared without any type.

Calls to the EXTERNAL procedure use standard Pascal syntax, and the Compiler checks that calls to the EXTERNAL agree in type and number of parameters with the EXTERNAL declaration. It is the user's responsibility to ensure that the assembly-language procedure respects the Pascal EXTERNAL declaration. The Linker checks only that the number of words of parameters agree between the Pascal and assembly-language declarations. For more information see the Apple Pascal Operating System Reference Manual.

The conventions of the surrounding system concerning register use and calling sequences must be respected by writers of assembly-language routines. On the Apple, all registers are available, and zero-page hexadecimal locations Ø through 35 are available as temporary variables. However, the Apple Pascal system also uses these locations as temporaries, so you should not expect data left there to be there when you execute the routine the next time. You can save variables in non-zero page memory by using the .BYTE or .WORD directives in your program to reserve space.

For assembly language functions (.FUNC's) the sequence is essentially the same, except that:

- Two words of zeros are pushed by the Compiler after any parameters are put on the stack.
- After the stack has been completely cleaned up at the routine exit time, the .FUNC must push the function result on the stack.

For an example of an EXTERNAL assembly-language procedure and an EXTERNAL assembly-language function, called from a Pascal program, see the example in the Apple Pascal Operating System Reference Manual. The EXTERNAL routine in that example is manually Linked into the calling program.

CHAPTER 6
OTHER DIFFERENCES

IDENTIFIERS

The underscore character _ is allowed in identifiers; however, the compiler ignores it. Therefore the identifiers

```
FIG_LEAF
FIGLEAF
```

are equivalent. (The Apple keyboard does not have the underscore character, but some external terminals do.)

CASE STATEMENTS

In Standard Pascal, if there is no case label equal to the value of the case selector, the result of the case statement is undefined. In Apple Pascal, if there is no case label matching the value of the case selector, then the next statement executed is the statement following the case statement.

COMMENTS

The Apple Pascal compiler recognizes any text appearing between either the symbols (* and *) or the symbols { and } as a comment. Text appearing between these symbols is ignored by the Compiler unless the first character of the comment is a dollarsign, in which case the comment is interpreted as a compiler option (see Chapter 4).

If the beginning of the comment is delimited by the (* symbol, the end of the comment must be delimited by the matching *) symbol, rather than the } symbol. When the comment begins with the { symbol, the comment continues until the matching } symbol appears. This feature allows you to "comment out" a section of a program which itself contains comments. This applies to external terminals only, since the only comment delimiter available on the Apple is the pair (* and *). An example of how the two kinds of comment delimiters are used on an external terminal:

```
{     XCP := XCP + 1;  (* ADJUST FOR SPECIAL CASE... *)    }
```

The compiler does not keep track of nested comments. When a comment symbol is encountered, the text is scanned for the matching comment symbol. The following text will result in a syntax error:

```
(* THIS IS A COMMENT  (* NESTED COMMENT *)  END OF FIRST COMMENT *)
                                              ^error here.
```

GOTO

Apple Pascal has a more limited form of GOTO statement than Standard Pascal. The destination of the GOTO statement must be in the same procedure as the COTO statement itself (considering the main program to be a procedure).

The compiler considers a COTO statement to be illegal unless the compiler option (*$G+*) is used; see Chapter 4.

PROGRAM HEADINGS

Although the Apple Pascal Compiler will permit a list of file parameters to be present following the program identifier (as in Standard Pascal), these parameters are ignored by the compiler and have no affect on the program being compiled.

SIZE LIMITS

The following is a list of maximum size limitations imposed upon the user by the current implementation of Apple Pascal:

- Maximum number of bytes of object code in a PROCEDURE or FUNCTION is 12ØØ. Local variables in a PROCEDURE or FUNCTION can occupy a maximum of about 18ØØØ words of memory.
- Maximum number of characters in a STRING variable is 255.
- Maximum number of elements in a SET is 32 * 16=512.
- Maximum number of segments a program can use is 16. This includes one segment for the main program, one for each SECMENT PROCEDURE and SEGMENT FUNCTION declared in the program, and one for each Regular UNIT that the program USES.
- Maximum number of PROCEDUREs or FUNCTIONs within a segment is 149.
- Maximum integer that can be represented is 32767; minimum is -32768.
- Maximum precision for REAL values is 32 bits.

EXTENDED COMPARISONS

Apple Pascal allows = and <> comparisons of arrays of exactly the same type and of record structures of exactly the same type. This can be

done without subscripting (in the case of arrays) or field identifiers (in the case of records). For example, given the declarations

```
VAR A: ARRAY[Ø..1Ø] OF INTEGER;
    B: ARRAY[Ø..1Ø] OF INTEGER;
```

then the following statement is legal:

```
IF A=B THEN ...
```

and the statement following the THEN will be executed if each element of A is equal to the corresponding element of B.

PROCEDURES AND FUNCTIONS AS PARAMETERS

Apple Pascal does not allow a PROCEDURE or FUNCTION to be declared as a formal parameter in the parameter list of another PROCEDURE or FUNCTION.

RECORD TYPES

There are two restrictions on record type declarations which are different from Standard Pascal syntax:

- A null field list is illegal; in other words the construction

  ```
  RECORD END;
  ```

  will cause an error.

- A null field within the parentheses of a variant field list is illegal; in other words a semicolon just before the closing parenthesis will cause an error.

THE ORD FUNCTION

The ORD function will accept a parameter of type POINTER, and return the numerical value of the pointer.

When the ORD function is given a BOOLEAN value as an actual parameter, the result is not always Ø or 1. It is most unlikely that a working program will ever encounter this situation, since there is little reason to take the ORD of a BOOLEAN value.

CHAPTER 7
SPECIAL UNITS SUPPLIED FOR THE APPLE

APPLE GRAPHICS: THE TURTLEGRAPHICS UNIT

This graphics package is called "Turtlegraphics" since it is based on the "turtles" devised by S. Papert and his coworkers at the Massachusetts Institute of Technology. To make graphics easy for children who might have difficulty understanding Cartesian coordinates, Papert et al. invented the idea of a "turtle" who could walk a given distance and turn through a specified angle while dragging a pen along. Very simple algorithms in this system (which could be called "relative polar coordinates") can give more interesting images than an algorithm of the same length in Cartesian coordinates.

Before any graphics can be used, they must be enabled by placing this declaration immediately after the program heading:

```
USES TURTLEGRAPHICS;
```

If this declaration appears, the graphics procedures and functions described in this section can be used. This declaration tells the Pascal system to get the graphics programs from the library. The SYSTEM.LIBRARY file must be on line when the program is R(un or eX(ecuted.

THE APPLE SCREEN

The Apple screen is a rectangle, with the origin (X=Ø,Y=Ø) at the LOWER LEFT corner. The upper right corner has the coordinates (X=279,Y=191). Since points may only be placed at integral coordinates, all arguments to the graphics functions are INTEGERs. (You can supply a REAL argument; it will be rounded to an INTEGER.)

There are two different screen images stored in the Apple's memory. One of them holds the text that you see when the computer is first turned on. The other holds a graphic image. There are three procedures that switch between the modes. They are INITTURTLE, TEXTMODE and GRAFMODE.

THE INITTURTLE PROCEDURE

This procedure has no parameters. It clears the screen, and allows the screen to be used for graphics rather than text. It is a good idea to use this routine before starting any graphics.

INITTURTLE does a few other things as well: the turtle (more about it later) is placed in the center of the screen facing right, the pen color is set to NONE (more about this later too) and the viewport is set to full screen (read on).

THE GRAFMODE PROCEDURE

This procedure has no parameters. It switches the monitor or TV to show the graphics screen, without the other initialization that INITTURTLE does. It is usually used to show graphics in a program that switches between graphics and text display.

THE TEXTMODE PROCEDURE

This procedure has no parameters. It switches from graphics mode (obtained by INITTURTLE or GRAFMODE) to showing text. When you switch to text mode, the image that you saw in GRAFMODE is not lost, but will still be there when you use GRAFMODE to go into graphics mode again (unless you deliberately changed it.) Upon termination of any program that uses graphics, the system automatically goes back into text mode.

THE VIEWPORT PROCEDURE

This procedure has the form

```
VIEWPORT (LEFT, RIGHT, BOTTOM, TOP)
```

where the four parameters give the boundaries you want the VIEWPORT to have. If you don't use this procedure, Apple Pascal assumes that you want to use the whole screen for your graphics.

There are occasions when it is handy to use only part of the screen, while safeguarding the rest from accidental use. For example, a small square near the middle of the screen might be selected as a viewport by the statement:

```
VIEWPORT (13Ø, 15Ø, 86, 1Ø6)
```

This example would allow the screen-plotting of all points whose X-coordinates are from 13Ø through 15Ø and whose Y-coordinates are from 86 through 1Ø6.

When a line is drawn using any of the graphic commands, it is automatically clipped so that only the portion which lies within the current viewport is displayed. Points whose coordinates are not in the current viewport, even those points that would not be on the screen at all, are legal but are ignored.

This allows some dramatic effects. It also allows you to plot off-screen all day, and never see a thing or get an error message. Clipping cannot be disabled.

USING COLOR: PENCOLOR

The PENCOLOR procedure sets the pen color. It has the form

```
PENCOLOR (COLOR)
```

The simplest colors are

```
WHITE

WHITE1 (two dots wide, for use with green and violet)

WHITE2 (two dots wide, for use with orange and blue)

BLACK

BLACK1 (two dots wide, for use with green and violet)

BLACK2 (two dots wide, for use with orange and blue)

GREEN

VIOLET

ORANGE

BLUE
```

If you'd like the drawing to be in GREEN, use the statement:

```
PENCOLOR (GREEN)
```

It may seem strange that aside from WHITE, BLACK, GREEN, VIOLET, ORANGE, and BLUE, there are two additional flavors of WHITE and BLACK. These are due to the intricate (not to say bizarre) way that color televisions produce their color, interacting with the technique that Apple uses to get a lot of color very economically. Rather than explaining how this all works, suffice it to say here that WHITE and BLACK give the finest lines possible, and the colors give a wider line in order to make the colors show. If you wish to make a white or black line that corresponds exactly in position and width with a green or violet line then you should use WHITE1 or BLACK1. If you wish to make a white or black line that corresponds exactly in position and width with an orange or blue line, then you should use WHITE2 or BLACK2.

On a black-and-white monitor or TV set, just use WHITE and BLACK.

The remaining colors are not really colors at all but are equally useful:

- NONE: Drawing with this "color" produces no change on the screen. It is useful for moving the turtle without drawing a line.

- REVERSE: Drawing with REVERSE changes BLACK to WHITE and WHITE to BLACK. It also changes WHITE1 to BLACK1, WHITE2 to BLACK2, GREEN to VIOLET and ORANGE to BLUE and vice versa. It is rather a magical "color". It allows you to draw, say, a line across a complex background and have it still show up.

- RADAR: This "color" has been left unused for future applications.

MORE COLOR: FILLSCREEN

The FILLSCREEN procedure has the form

```
FILLSCREEN (COLOR)
```

FILLSCREEN fills the entire viewport with the specified color. For example

```
FILLSCREEN (BLACK)
```

clears the viewport. The statement

```
FILLSCREEN (REVERSE)
```

makes a "negative" of the contents of the viewport.

When you invoke TURTLEGRAPHICS, a new variable type called SCREENCOLOR is automatically created. It is defined as follows:

```
SCREENCOLOR = (NONE, WHITE, BLACK, REVERSE, RADAR, BLACK1, GREEN,
               VIOLET, WHITE1, BLACK2, ORANGE, BLUE, WHITE2);
```

SCREENCOLOR has all the usual characteristics of a Pascal type. It is useful when you declare a variable that will be used to store a color.

TURTLE GRAPHIC PROCEDURES: TURNTO, TURN, AND MOVE

At last we're back to the imaginary turtle. Initially, the turtle sits at the center of the screen, facing right. The turtle can only do two things: it can turn or it can walk in the direction it is facing. As it walks, it leaves behind a trail of ink (!) in the current pen color.

The TURNTO procedure has the form

```
TURNTO (DEGREES)
```

where DEGREES is an integer which is treated modulo 360; thus its effective value is in the range -359 through 359. When invoked, this procedure causes the turtle to turn from its present angle to the indicated angle. 0 is exactly to the right, and counterclockwise rotation represents increasing angles. This command never causes any change to the image on the screen.

The TURN procedure has the form

```
TURN (DEGREES)
```

where DEGREES is again an integer which is treated modulo 360; thus its effective value is in the range -359 through 359. This procedure causes the turtle to rotate counterclockwise from its current direction through the specified angle. It causes no change to the image on the screen.

The MOVE procedure has the form

```
MOVE (DISTANCE)
```

where DISTANCE is an integer. This procedure makes the turtle move IN THE DIRECTION IN WHICH IT IS POINTING a distance given by the integer DISTANCE. It leaves a trail in the current pen color. The sequence of statements:

```
PENCOLOR (WHITE);
MOVE (50);
TURN (120);
MOVE (50);
TURN (120);
MOVE (50)
```

draws an equilateral triangle, for instance.

TURTLE GRAPHIC FUNCTIONS: TURTLEX, TURTLEY, TURTLEANG, AND SCREENBIT

These functions allow you to interrogate the computer about the current state of the turtle and the screen.

The TURTLEX and TURTLEY functions (no parameters) return integers giving the current X and Y coordinates of the turtle.

The TURTLEANG function (no parameters) returns an integer giving the current turtle angle as a positive number of degrees. Note that if you use TURNTO and then TURTLEANG, the value returned by TURTLEANG may not be the same value you gave with TURNTO. For example, after

```
TURNTO(-90)
```

TURTLEANG will return 270, not -90.

The SCREENBIT function has the form

```
SCREENBIT (X,Y)
```

where X and Y are screen coordinates. This function returns the BOOLEAN value TRUE if the specified location on the screen is not black, and FALSE if it is black. It doesn't tell you what color is at that point, but only whether there is something non-black there or not.

CARTESIAN GRAPHICS: THE MOVETO PROCEDURE

Earlier we said that in turtle graphics, the turtle can only walk in the direction it is facing. But in Cartesian graphics, the turtle can move to a specified point on the screen without turning. The MOVETO procedure has the form

```
MOVETO (X, Y)
```

where X and Y are screen coordinates. MOVETO moves the turtle to the point (X,Y). This creates a line in the current pen color from the turtle's last position to the point (X,Y).

The direction of the turtle is not changed by MOVETO.

GRAPHIC ARRAYS: THE DRAWBLOCK PROCEDURE

The DRAWBLOCK procedure has the form

```
DRAWBLOCK (SOURCE, ROWSIZE, XSKIP, YSKIP, WIDTH, HEIGHT,
           XSCREEN, YSCREEN, MODE)
```

where the SOURCE parameter is the name (without subscripts) of a variable which should be a two-dimensional PACKED ARRAY OF BOOLEAN (see note below). All the other parameters are integers.

DRAWBLOCK treats each BOOLEAN element of SOURCE as a "dot" -- true for white or false for black. It copies the array of "dots" (or a portion of it) from memory onto the screen to form a screen image. The first dimension of the array is the number of rows in the array; the second dimension is the number of elements in each row.

You may choose to copy the entire SOURCE array, or you may choose to copy any specified "window" from the array, using only those dots in the array from XSKIP to XSKIP+WIDTH and from YSKIP to YSKIP+HEIGHT. Furthermore, you can specify the starting screen position for the copy, at (XSCREEN, YSCREEN).

- SOURCE is the name of the two-dimensional PACKED ARRAY OF BOOLEAN to be copied (see note below).

- SIZE is the number of bytes (not dots) per row in the array. You can calculate this from the formula

  ```
  2*( (X+15) DIV 16 )
  ```

  where X is the number of dots in each row.

- XSKIP tells how many horizontal dots in the array to skip over before the copying process is started.

- YSKIP tells how many vertical dots in the array to skip over before beginning the copying process. Note that copies are made starting from the bottom up -- i.e. the first row copied from the array is the bottom row of the screen copy.

- WIDTH tells how many dots' width of the array, starting at XSKIP, will be used.

- HEIGHT tells how many dots' height of the array, starting at YSKIP, will be used.

- XSCREEN and YSCREEN are the coordinates of the lower left corner of the area to be copied into. The WIDTH and HEIGHT determine the size of the rectangle.

- MODE ranges from Ø through 15. The MODE determines what appears on the portion of the screen specified by the other parameters. It is quite a powerful option, which can simply send white or black to the screen, irrespective of what is in the array, copy the array literally, or combine the contents of the array and the screen and send the result to the screen. The following table specifies what operation is performed on the data in the array and on the screen, and thus what appears on the screen. (The logical notation uses A for the array, and S for the screen. The symbol ~ means NOT.)

MODE	Effect
Ø	Fills area on screen with black.
1	NOR of array with screen. (A NOR S)
2	AND of array with complement of screen. (A AND ~S)
3	Complements area on screen. (~S)
4	AND of complement of array with screen. (~A AND S)
5	Complements the array. (~A)
6	XOR of array with screen. (A XOR S)
7	NAND of array with screen. (A NAND S)
8	AND of array with screen. (A AND S)
9	EQUIVALENCE of array with screen. (A = S)
1Ø	Copies array to screen. (A)
11	OR of array with complement of screen. (A OR ~S)
12	Screen replaces screen. (S)
13	OR of complement of array with screen. (~A OR S)
14	OR of array with screen. (A OR S)
15	Fills area on screen with white.

The demonstration program GRAFDEMO.TEXT, on APPLE3:, contains many examples of how to use the turtlegraphics routines. In particular, procedures BUTTER1, etc., give strings to procedure STUFF, which converts them to a PACKED ARRAY OF BOOLEAN named BUTTER; and procedure FLUTTER uses the DRAWBLOCK routine to display the array BUTTER on the screen.

Actually, the SOURCE parameter can be of any type except a FILE type; DRAWBLOCK really deals with an array of bits in memory which begins at the address of SOURCE and whose size and organization depend on the other parameters. For example, the following procedure uses a single BOOLEAN variable instead of an array. The procedure plots a single dot on the screen at specified coordinates (X,Y):

```
PROCEDURE PLOTDOT(X, Y: INTEGER);
  VAR DOT:BOOLEAN;
  BEGIN
    DRAWBLOCK(DOT,1,Ø,Ø,1,1,X,Y,3)
  END;
```

However, for most programs the most convenient way to handle the array is to use a two-dimensional PACKED ARRAY OF BOOLEAN as described previously.

TEXT AS GRAPHICS: WCHAR, WSTRING, AND CHARTYPE

Three procedures allow you to put characters on the graphics screen. If the turtle is at (X,Y) you can use these procedures to put a character or string on the screen with its lower left corner at (X,Y). Each character occupies a rectangular area 7 dots wide and 8 dots high on the screen.

These procedures use an array stored in the file SYSTEM.CHARSET on diskette APPLE1:. This array contains all the characters used, and is read in by the initialization routine when your program USES TURTLEGRAPHICS. If you make up an array containing your own character set, you should rename the old SYSTEM.CHARSET and then name your new array SYSTEM.CHARSET (see note at the end of this chapter).

WSTRING and WCHAR use the procedure DRAWBLOCK to copy each character from the array onto the screen. The MODE parameter that they use is set by the CHARTYPE procedure.

The WCHAR procedure has the form

```
WCHAR (CH)
```

where CH is a an expression of type CHAR. This procedure places the character on the screen with its lower left corner at the current location of the turtle. When this procedure is used, the turtle is shifted to the right 7 dots from its old position. For example, this puts an X in the center of the screen:

```
PENCOLOR (NONE);
MOVETO (137,9Ø);
WCHAR ('X')
```

In this example, note that it was not necessary to specify a new pen color before calling WCHAR. The character is not plotted with the current pen color; rather it depends on the current MODE, just as DRAWBLOCK does. For details, see CHARTYPE below.

The CHAR value passed to WCHAR is restricted to the first 128 characters of the ASCII set as shown in Table 7 of Appendix B.

The WSTRING procedure has the form

```
WSTRING (S)
```

where S is an expression of type STRING. An entire string of characters is placed on the screen with the lower left corner of the first character at the current turtle position. The turtle is shifted 7 dots to the right for each character in the string. This procedure calls WCHAR for each character in the string.

The characters in the STRING value passed to WSTRING are restricted to the first 128 characters of the ASCII set as shown in Table 7 of Appendix B.

The CHARTYPE procedure has the form

```
CHARTYPE (MODE)
```

where MODE is an integer selecting one of the 16 MODEs described above for DRAWBLOCK. MODE defines the way characters get written on the screen; it works for WCHAR and WSTRING just as it works for DRAWBLOCK.

The default MODE is 1Ø, which places each character on the screen in white, surrounded by a black rectangle. MODE 5 is the inverse of MODE 1Ø: each character is in black surrounded by a white rectangle.

One of the most useful other MODEs is 6, which does an exclusive OR of the character with the current contents of the screen. If you use MODE 6 to draw a character or string and then redraw it at the same location with MODE 6, the effect is to erase the character or string, leaving the original image unaffected. This is especially useful for user messages in a graphics-oriented program.

If you wish to create your own character set file for use with WCHAR and WSTRING, it must be structured as follows:

- The file consists of 1Ø24 bytes.
- Starting with the first byte in the file, each character in the character set is represented by 8 contiguous bytes.
- Each byte represents one row of 8 dots in the character image. The first byte of each character representation is the bottom row of the image.

- The least significant bit of each byte is the leftmost dot in the row.

- The most significant bit of each byte is ignored; the rows are only seven dots each.

Such a file can be created either in assembly language or in Pascal. In Pascal, you can build the character representations in memory as packed arrays of the type Ø..255 since each element of such an array is in effect a byte. For example, you might use the declarations

```
TYPE CHARIMAGE=PACKED ARRAY[Ø..7] OF Ø..255;
     CHARSET=PACKED ARRAY[Ø..127] OF CHARIMAGE;
     CHARFILE=FILE OF CHARSET;

VAR CHARACTERS:CHARSET;
    OUTFILE:CHARFILE;
```


OTHER SPECIAL APPLE FEATURES: THE APPLESTUFF UNIT

This section tells you how to generate random numbers, how to use the game paddle and button inputs, how to read the cassette audio input, how to switch the game-control's TTL outputs and how to generate sounds on the Apple's speaker. To use these special Apple features from Pascal, you first have to place the declaration

```
USES APPLESTUFF;
```

immediately after the program heading. If you wish to use both turtle graphics and the Apple features you would say

```
USES TURTLEGRAPHICS, APPLESTUFF;
```

since there can only be one USES in a program.

THE RANDOM FUNCTION

RANDOM is an integer function with no parameters. It returns a value from Ø through 32767. If RANDOM is called repeatedly, the result is a psuedo-random sequence of integers. The statement

```
WRITELN (RANDOM)
```

will display an integer between the indicated limits.

A typical application of this function is to get a pseudo-random number, say, between LOW and HIGH inclusive. The expression

```
LOW + RANDOM MOD (HIGH-LOW+1)
```

is sometimes used where results are not critical, but the values formed by this expression are <u>not</u> evenly distributed over the range LOW

through HIGH. If you want pseudo-random integers evenly distributed over a range, you can use the following function:

```
FUNCTION RAND (LOW, HIGH:INTEGER; VAR ERROR:BOOLEAN):INTEGER;
  VAR MX, C, D: INTEGER;
  BEGIN
    RAND := Ø;
    ERROR := TRUE;
    IF LOW > HIGH THEN EXIT(RAND); (*error exit*)
    IF LOW <= Ø THEN
      IF HIGH > MAXINT + LOW THEN EXIT(RAND); (*error exit*)

    ERROR := FALSE; (*no errors*)
    IF LOW = HIGH THEN RAND := LOW
                  ELSE BEGIN
                         C := HIGH - LOW + 1;
                         MX := (MAXINT - HIGH + LOW) DIV C + 1;
                         MX := MX * (HIGH - LOW) + (MX - 1);
                         REPEAT D := RANDOM UNTIL D <= MX;
                         RAND := LOW + D MOD C
                       END
  END;
```

If HIGH is not greater than LOW, or the difference between HIGH and LOW would exceed MAXINT, RAND returns Ø and sets the ERROR parameter to true. Otherwise, RAND returns evenly distributed pseudo-random integer values between LOW and HIGH (inclusive).

THE RANDOMIZE PROCEDURE

RANDOMIZE is a procedure with no parameters. Each time you run a given program using RANDOM, you will get the same random sequence unless you use RANDOMIZE.

RANDOMIZE uses a time-dependent location to generate a starting point for the random number generator. The starting point changes each time you do any input or output operation in your program. If you use no I/O, the starting point for the random sequence does not change.

THE KEYPRESS FUNCTION

This function, which has no parameters, returns true if a key has been pressed on the keyboard since the program started or since the last time the keyboard was read (whichever is most recent). KEYPRESS does not read the character from CONSOLE or KEYBOARD or have any other effect on I/O. The statement

```
IF KEYPRESS THEN READ(KEYBOARD, CH)
```

(where CH is a CHAR variable) has the effect of reading the last character typed on the keyboard. This could be used to retrieve a character typed while the program was doing something else -- for instance, displaying graphics.

Once KEYPRESS becomes true it remains true until a GET, READ, or READLN accesses either the INPUT file or the KEYBOARD file, or until a UNITREAD accesses the keyboard device.

KEYPRESS does not work with an external terminal connected via a serial interface card. It will always return FALSE with such a terminal.

PADDLE, BUTTON, AND TTLOUT

The PADDLE function has the form

```
PADDLE (SELECT)
```

where SELECT is an integer treated modulo 4 to select one of the four paddle inputs numbered Ø, 1, 2, and 3. PADDLE returns an integer in the range Ø to 255 which represents the position of the selected paddle. A 15ØK variable resistance can be connected in place of any of the four paddles.

If you try to read two paddles too quickly in succession, e.g.

```
WRITELN (PADDLE (Ø));
WRITELN (PADDLE (1))
```

the hardware will not be able to keep up. A suitable delay is given by the loop

```
FOR I := Ø TO 3 DO;
```

The BUTTON function has the form

```
BUTTON (SELECT)
```

where SELECT is an integer treated modulo 4 to select one of the three button inputs numbered Ø, 1, and 2, or the audio cassette input numbered 3. The BUTTON function returns a BOOLEAN value of TRUE if the selected game-control button is pressed, and FALSE otherwise.

When BUTTON(3) is used to read the audio cassette input, it samples the cassette input which changes from TRUE to FALSE and vice versa at each zero crossing of the input signal.

There are four TTL level outputs available on the game connector along with the button and paddle inputs. The TTLOUT procedure is used to turn these outputs on or off. TTLOUT has the form

```
TTLOUT (SELECT, DATA)
```

where SELECT is an integer treated modulo 4 to select one of the four TTL outputs numbered Ø, 1, 2, and 3. DATA is a BOOLEAN expression.

If DATA is TRUE, then the selected output is turned on. It remains on until TTLOUT is invoked with the DATA set to FALSE.

MAKING MUSIC: THE NOTE PROCEDURE

The NOTE procedure has the form

```
NOTE (PITCH, DURATION)
```

where PITCH is an integer from Ø through 5Ø and DURATION is an integer from Ø through 255.

A PITCH of Ø is used for a rest, and 2 through 48 yield a tempered (approximately) chromatic scale. DURATION is in arbitrary units of time.

NOTE (1,1) gives a click.

A chromatic scale is played by the following program:

```
PROGRAM SCALE;

USES APPLESTUFF;
VAR PITCH, DURATION: INTEGER;

  BEGIN

    DURATION := 1ØØ;
    FOR PITCH := 12 TO 24 DO
      NOTE (PITCH, DURATION)

  END.
```


TRANSCENDENTAL FUNCTIONS: THE TRANSCEND UNIT

In Apple Pascal, the transcendental functions are not built into the language. To use this set of functions you must place the declaration

```
USES TRANSCEND;
```

immediately after the PROGRAM heading. If you wish to use, say, APPLESTUFF with the transcendental functions, you would write

```
USES TRANSCEND, APPLESTUFF;
```

All ANGLE and NUMBER arguments are real, and the ANGLE arguments are in radians. All of these functions return real values, and values returned by the ATAN function are in radians. The following functions are provided:

```
SIN (ANGLE)

COS (ANGLE)

EXP (NUMBER)

ATAN (NUMBER)          (Note: this is the same function
                         as Standard Pascal's ARCTAN)

LN (NUMBER)

LOG (NUMBER)

SQRT (NUMBER)
```

APPENDIX A
DEMONSTRATION PROGRAMS

INTRODUCTION

This appendix presents a graphics program which is fully annotated, both by a narrative explanation and by copious comments in the source text. This program is followed by commentaries on the demonstration programs supplied with Apple Pascal.

A word of caution is in order regarding all of these programs. They are presented so as to give you examples that you can run without any modification, and also study the source text to see how it works. They are not intended to be models of the best possible programming technique; that would be entirely beyond the scope of this manual. They do work, and they do demonstrate ways of doing certain useful things in Apple Pascal. With this caution in mind, use the programs as learning tools. One of the best ways to learn might be to try introducing modifications into one of them.

A FULLY ANNOTATED GRAPHICS PROGRAM

The following demonstration program, PATTERNS, is intended to illustrate some helpful points about Apple Pascal. The program creates pleasant graphics by drawing a triangle on the screen and then repeatedly rotating it by a few degrees and redrawing it. The points of the triangle are always on the edge of an invisible circle of radius 95 (which fills the height of the screen) but apart from that it is a random triangle. The angle by which it is rotated each time it is drawn is also random, though it is always between 3 and 15 degrees.

The color used to draw the triangle is REVERSE, which has intriguing effects when one image is drawn over another and the lines intersect at small angles. Also, as the triangle is repeatedly rotated and redrawn a circular pattern is built up; but eventually the triangle gets rotated back to its original position. When this happens, each new image is exactly superimposed on an old one. Because of the REVERSE color, this erases the old image! When all the old images have been erased, the program clears the screen, generates a new triangle with a different shape, and starts all over.

This repetition continues until the user signals it to halt by pressing any key. The KEYPRESS function, in the APPLESTUFF unit, can be used to find out whether the user has pressed a key. (KEYPRESS is described in Chapter 7.)

The program is given in full, with comments, at the end of this appendix. What follows is a description of how a program like this can be developed. Of course, in real life there are mistakes and false starts. Here, for the sake of learning some principles, we pretend that the development of the program proceeds without a hitch.

This is a fairly complicated program, so we will develop it in sections. First we can write a sketchy outline of the program:

```
BEGIN
  REPEAT
    (*Create a random triangular pattern*);
    THETA:=(*random number from 3 to 15*)
    REPEAT
      (*Rotate the triangle, using the angle THETA*);
      (*Draw the rotated triangle on the screen*)
    UNTIL (*Complete pattern has been erased*)
  UNTIL KEYPRESS
END.
```

To fill in this outline, we begin with a program heading, a USES declaration, some useful constants, two variable declarations, and a skeleton of the inner REPEAT statement:

```
PROGRAM PATTERNS;
USES TURTLEGRAPHICS,APPLESTUFF;

CONST MAXX=28Ø; MAXY=191; (*Maximum X and Y coordinates*)
      RADIUS=95; (*Radius of pattern*)

VAR CYCLES:Ø..2;
    THETA:3..15;

BEGIN
  REPEAT
    (*Create a random triangular pattern*);
    THETA:=(*random number from 3 to 15*);
    CYCLES:=Ø
    REPEAT
      (*Rotate the triangle, using the angle THETA*);
      PENCOLOR(REVERSE);
      (*Draw the rotated triangle on the screen*);
      IF (*the rotated triangle matches the original triangle*)
        THEN CYCLES:=CYCLES+1
    UNTIL CYCLES=2
  UNTIL KEYPRESS
END.
```

The variable CYCLES is a counter for the number of times the triangle has been rotated back to its original position. When CYCLES=1, the circular pattern begins to be erased because each new triangle is drawn in the REVERSE color on top of a previous triangle. When CYCLES=2, the entire pattern has been erased.

We can now begin replacing comments with actual statements. For example, we already have a variable, THETA, which is the number of degrees to rotate the pattern. So it is natural to replace the first comment in the inner REPEAT with a call to a procedure named ROTATE which takes an INTEGER parameter. The value used for the parameter

will be the variable THETA. ROTATE will need to be declared; now we have

```
...

PROCEDURE ROTATE(ANGLE:INTEGER);
  (*To be completed...*)

BEGIN
  REPEAT
    (*Create a random triangular pattern*);
    THETA:=(*random number from 3 to 15*);
    CYCLES:=Ø
    REPEAT
      ROTATE(THETA);
...
```

To draw the triangle on the screen, we must first consider how the triangle is represented in memory. We can think of the triangle as three points; how shall we represent a point? A point can be represented by two numbers -- an X and a Y coordinate. Therefore we can define a type POINT, as shown below. Then we can represent the triangle as an array named TRGL, of type POINT. We will also declare a variable C to use as an index for the array TRGL.

```
...

TYPE POINT=RECORD X:Ø..MAXX;
                  Y:Ø..MAXY
           END;

VAR CYCLES:Ø..2;
    THETA:3..15;
    TRGL:ARRAY[1..3] OF POINT;
    C:1..3;

...
```

Assuming that the ROTATE procedure leaves the rotated coordinates in the array TRGL and that it leaves the turtle at the third corner of the triangle, we can use Cartesian graphics to draw the new triangle:

```
        ...

        PENCOLOR(REVERSE);
        FOR C:=1 TO 3 DO MOVETO(TRGL[C].X, TRGL[C].Y);

        ...
```

The remaining comment in the inner REPEAT statement calls for testing whether the rotated triangle matches the original one. To achieve this, assume that when the triangle is first created the coordinates of the third corner are saved in a variable named CORNER. Now we need only test as follows:

```
        ...

        IF TRGL[3]=CORNER THEN CYCLES:=CYCLES+1

        ...
```

At this point, the program is as follows:

```
PROGRAM PATTERNS;
USES TURTLEGRAPHICS,APPLESTUFF;

CONST MAXX=28Ø; MAXY=191; (*Maximum X and Y coordinates*)
      RADIUS=95; (*Radius of pattern*)

TYPE POINT=RECORD X:Ø..MAXX;
                  Y:Ø..MAXY
           END;

VAR CYCLES:Ø..2;
    THETA:3..15;
    TRGL:ARRAY[1..3] OF POINT;
    C:1..3;
    CORNER:POINT;

PROCEDURE ROTATE(ANGLE:INTEGER);
  (*To be completed; must leave new corner coordinates
    in TRGL and leave turtle at third corner.*)

BEGIN
  REPEAT
    (*Create a random triangular pattern*);
    THETA:=(*random number from 3 to 15*);
    CYCLES:=Ø
    REPEAT
      ROTATE(THETA);
      PENCOLOR(REVERSE);
      FOR C:=1 TO 3 DO MOVETO(TRGL[C].X, TRGL[C].Y);
      IF TRGL[3]=CORNER THEN CYCLES:=CYCLES+1
    UNTIL CYCLES=2
  UNTIL KEYPRESS
END.
```

The inner REPEAT statement will repeatedly rotate the triangle and draw it, using the REVERSE color, building up a circular pattern on the screen and then erasing it by drawing over it. When the pattern has been erased, the inner REPEAT terminates.

Now we can begin filling in the outer REPEAT. The statements added to the outer REPEAT require another procedure, MAKETRGL, and a function, ARBITRARY.

```
...

FUNCTION ARBITRARY(LOW, HIGH:INTEGER):INTEGER;
  (*To be completed; must return an integer value in the
    range LOW..HIGH.*)

PROCEDURE MAKETRGL;
  (*To be completed; must leave corner coordinates in TRGL
    and also initialize CORNER with coordinates of third
    corner.*)

BEGIN
  REPEAT
    MAKETRGL;                  (*Make triangular pattern*)
    THETA:=ARBITRARY(3, 15); (*Choose angle for rotating triangle*)
    CYCLES:=Ø;                 (*Clear the cycle counter*)
    REPEAT
      ROTATE(THETA);
      PENCOLOR(REVERSE);
      FOR C:=1 TO 3 DO MOVETO(TRGL[C].X, TRGL[C].Y);
      IF TRGL[3]=CORNER THEN CYCLES:=CYCLES+1
    UNTIL CYCLES=2
  UNTIL KEYPRESS
END.
```

The outer REPEAT first calls MAKETRGL. This procedure, still to be defined, chooses three random points on a circle of radius 95 and stores their coordinates in the array TRGL. It also stores the coordinates of the third corner in the variable CORNER.

Next, the function ARBITRARY is used to assign a random value to THETA, the number of degrees to rotate the triangle.

The main program is nearly complete. It remains only to add one new variable named CENTER (of type POINT), and a few initializing statements before the outer REPEAT:

```
...

VAR CYCLES:Ø..2;
    THETA:3..15;
    TRGL:ARRAY[1..3] OF POINT;
    C:1..3;
    CORNER:POINT;
    CENTER:POINT;

BEGIN
  RANDOMIZE;              (*To get a different sequence each time
                            program is executed*)
  INITTURTLE;             (*Always do this to use TURTLEGRAPHICS*)
  CENTER.X:=TURTLEX;      (*The turtle is at the center because
  CENTER.Y:=TURTLEY;        INITTURTLE leaves it there.  Save
                            its coordinates in CENTER.*)
  REPEAT
    ...

    REPEAT
      ...

    UNTIL CYCLES=2
  UNTIL KEYPRESS
END.
```

The main program is complete, and now we must define the two procedures MAKETRGL and ROTATE and the function ARBITRARY.

The ARBITRARY function is shown in the complete program at the end of this appendix. It is a simplified version of the RAND function given in Chapter 7, in the discussion of the built-in function RANDOM.

RAND handles unacceptable parameters by setting a VAR parameter of type BOOLEAN. ARBITRARY does not need this error-handling capability since it will always be called with constants as parameters. Similarly, RAND has a special provision for the case where the HIGH and LOW parameters are equal; ARBITRARY does not have this provision, and HIGH must be strictly greater than LOW.

In other respects, ARBITRARY is the same as RAND. Incidentally, the complexity of the calculation in both versions is due to the fact that two numbers cannot be added or subtracted if the result would exceed the value MAXINT (32767). The function has to get around this limitation by using the intermediate value MX.

The MAKETRGL procedure must choose three random points on a circle of radius 95, with its center at CENTER. To select three random points, the following method is used:

```
VAR I:1..3;

...

FOR I:=1 TO 3 DO BEGIN
(*Move the turtle to the CENTER point:*)
  MOVETO(CENTER.X, CENTER.Y);

(*Select a random direction to move the turtle away from CENTER,
  and store this angle in an array named DIRECTION; this array will
  need to be declared:*)
  DIRECTION[I]:=ARBITRARY(Ø,359);

(*Turn the turtle in the selected direction:*)
  TURNTO(DIRECTION[I]);

(*Move out to the edge of the circle:*)
  MOVE(RADIUS);

(*Store the turtle's coordinates in the TRGL array:*)
  TRGL[I].X:=TURTLEX;
  TRGL[I].Y:=TURTLEY
END
```

The DIRECTION array will be used by the ROTATE procedure, so it will need to be declared at the beginning of the program -- not within the MAKETRGL procedure.

Since we don't want to draw anything at this point, we set the color to NONE before starting the FOR statement. After three times through the FOR statement, the turtle is at the third corner of the triangle, so we save its position in the CORNER variable for use in the main program. The complete procedure is

```
PROCEDURE MAKETRGL;
  VAR I:1..3;
  BEGIN
    PENCOLOR(NONE);
    FOR I:=1 TO 3 DO BEGIN
      MOVETO(CENTER. , CENTER.Y);
      DIRECTION[I]:=ARBITRARY(Ø, 359);
      TURNTO(DIRECTION[I];
      MOVE(RADIUS);
      TRGL[I].X:=TURTLEX;
      TRGL[I].Y:=TURTLEY
    END;
    CORNER.X:=TURTLEX;
    CORNER.Y:=TURTLEY
  END;
```


The ROTATE procedure works very much like the MAKETRGL procedure, but instead of using random angles it uses the angles found in the DIRECTION array -- after adding ANGLE to each of them and taking the result MOD 36Ø. It stores the resulting points in the TRGL array, but does not change CORNER. The effect is to replace each point in TRGL with a new point created by rotation through ANGLE degrees. The complete ROTATE procedure is

```
PROCEDURE ROTATE(ANGLE:INTEGER);
  VAR I:1..3;
  BEGIN
    PENCOLOR(NONE);
    FOR I:=1 TO 3 DO BEGIN
      MOVETO(CENTER.X, CENTER.Y);
      DIRECTION[I]:=(DIRECTION[I]+ANGLE) MOD 36Ø;
      TURNTO (DIRECTION[I]);
      MOVE(RADIUS);
      TRGL[I].X:=TURTLEX;
      TRGL[I].Y:=TURTLEY
    END
  END;
```

Note that the MOD 36Ø operation is necessary because if the program ran for a long time, the result of DIRECTION[I]+ANGLE could eventually exceed MAXINT and cause a run-time error.

All that remains is to declare the array DIRECTION:

```
DIRECTION:ARRAY[1..3] OF INTEGER;
```

The complete program begins on the following page.

```
PROGRAM PATTERNS;
USES TURTLEGRAPHICS,APPLESTUFF;

(*************************************************************************)
CONST
(*Maximum X and Y coordinates*)
     MAXX=28Ø; MAXY=191;
(*Radius of pattern*)
     RADIUS=95;

(*************************************************************************)
TYPE
(*This type stores one set of screen coordinates*)
    POINT=RECORD X:Ø..MAXX;
                 Y:Ø..MAXY
          END;

(*************************************************************************)
VAR
(*Counter for how many times triangle has been rotated back to its
initial position*)
    CYCLES:Ø..2;
(*Angle for rotating triangle*)
    THETA:3..15;
(*Array to store coordinates of corners of triangle*)
    TRGL:ARRAY[1..3] OF POINT;
(*Index for corners of triangle*)
    C:1..3;
(*Point to store coordinates of one corner of triangle, before any
  rotations*)
    CORNER:POINT;
(*Point to store coordinates of center of screen*)
    CENTER:POINT;
(*Array to store direction angles used to generate triangle*)
    DIRECTION:ARRAY[1..3] OF INTEGER;

(*************************************************************************)
FUNCTION ARBITRARY (LOW, HIGH:INTEGER):INTEGER;

(*Returns a pseudo-random integer in the range LOW through HIGH.  This
function should only be called with constants as parameters.  HIGH must
be strictly greater than LOW; it must not be equal to LOW.  Also the
difference between HIGH and LOW must not exceed MAXINT.*)

  VAR MX, Z, D: INTEGER;
  BEGIN
    Z:=HIGH-LOW+1;
    MX:=(MAXINT-HIGH+LOW) DIV Z+1;
    MX:=MX*(HIGH-LOW)+(MX-1);
    REPEAT D:=RANDOM UNTIL D <= MX;
    ARBITRARY:=LOW+D MOD Z
  END;
```



```
(*************************************************************************)
PROCEDURE MAKETRGL;

(*Make a triangle, defined by three randomly chosen points at a distance
RADIUS from the point CENTER.  Choose each point by starting at CENTER,
turning to a random angle, and moving the distance RADIUS.  Store the
angles in DIRECTION, the point coordinates in TRGL, and the third point
(for future reference) in CORNER.  Notice how conveniently this is done
by moving the turtle around with the color NONE.*)

  VAR I:1..3;
  BEGIN
    PENCOLOR(NONE);
    FOR I:=1 TO 3 DO BEGIN
      MOVETO(CENTER.X, CENTER.Y);
      DIRECTION[I]:=ARBITRARY(Ø, 359);
      TURNTO(DIRECTION[I]);
      MOVE(RADIUS);
      TRGL[I].X:=TURTLEX;
      TRGL[I].Y:=TURTLEY
    END;
    CORNER.X:=TURTLEX;
    CORNER.Y:=TURTLEY
  END;

(*************************************************************************)
PROCEDURE ROTATE(ANGLE:INTEGER);

(*Rotate the triangle defined by point coordinates in TRGL and angles in
DIRECTION, by adding ANGLE to the angles in DIRECTION, taking the
result MOD 36Ø, and using these angles to determine the new corner
coordinates.  Again the turtle is moved around using the color NONE.*)

  VAR I:1..3;
  BEGIN
    PENCOLOR(NONE);
    FOR I:=1 TO 3 DO BEGIN
      MOVETO(CENTER.X, CENTER.Y);
      DIRECTION[I]:=(DIRECTION[I]+ANGLE) MOD 36Ø;
      TURNTO(DIRECTION[I]);
      MOVE(RADIUS);
      TRGL[I].X:=TURTLEX;
      TRGL[I].Y:=TURTLEY
    END
  END;
```

```
(*********************************************************************)

                          (*Main Program*)

BEGIN

(*Do initializations that will not need to be repeated*)

  RANDOMIZE;             (*To get a different sequence each time
                           program is executed*)
  INITTURTLE;            (*Always do this to use TURTLEGRAPHICS*)
  CENTER.X:=TURTLEX;     (*The turtle is at the center because
                           INITTURTLE leaves it there.  Save its
                           coordinates in CENTER.*)
  CENTER.Y:=TURTLEY;

(*The following (outer) REPEAT statement creates a new triangular
pattern each time through.*)

  REPEAT
    MAKETRGL;                 (*Make triangular pattern*)
    THETA:=ARBITRARY(3, 15); (*Choose angle for rotating triangle*)
    CYCLES:=Ø;                (*Clear the cycle counter*)
```



```
(*The following (inner) REPEAT statement draws the triangle in a new
rotated position each time through.*)

    REPEAT

(*Rotate the triangle.*)

      ROTATE(THETA);

(*Draw the triangle.  This is conveniently done with Cartesian
graphics, since the coordinates are all set up.*)

      PENCOLOR(REVERSE);
      FOR C:=1 TO 3 DO MOVETO(TRGL[C].X, TRGL[C].Y);

(*Now, if the third corner of the triangle matches the CORNER value
saved earlier (by MAKETRGL), then the triangle has been rotated back to
its original position.*)

      IF TRGL[3]=CORNER THEN CYCLES:=CYCLES+1

(*End the repetition if the triangle has returned to its original
position twice.  When this is the case, the pattern has been erased by
being drawn over with the REVERSE color.*)

    UNTIL CYCLES=2

(*End the outer REPEAT statement when a key is pressed.*)

  UNTIL KEYPRESS

END.
```

OTHER DEMONSTRATION PROGRAMS

A set of demonstration programs is supplied with the Pascal System. Although these programs are not fully annotated, they are worth careful study by any student of Pascal. The following are brief descriptions of the programs.

The .TEXT version of each program has been included on diskette APPLE3: so that you can read the program's text into the Editor, to see how the program was written and to try modifications of your own.

DISKETTE FILES NEEDED

The following diskette files allow you to execute the various demonstration programs. The notation xxxxxx stands for the name of a particular demonstraion program.

xxxxxx.CODE	(any diskette, any drive)
SYSTEM.LIBRARY	(boot diskette, boot drive)
SYSTEM.CHARSET	(any diskette, any drive; required if WCHAR or WSTRING used)

One-drive note: Use the Filer to T(ransfer the desired demonstration program's .CODE file to your boot diskette, APPLE0: or APPLE1:. Then you can X(ecute the program with the boot diskette in the disk drive.

Multi-drive note: You should place your boot diskette, APPLE0: or APPLE1: , in the boot drive. The demonstration programs are all normally found on diskette APPLE3:. With APPLE3: in any available disk drive, you are ready to X(ecute the demonstration programs.

If you just wish to examine the text version of a demonstration program, there are two ways to proceed:

- For a quick look, put diskette APPLE3: in any available drive, and then use the Filer to T(ransfer the desired program's .TEXT file from APPLE3: to CONSOLE:. To stop the program's listing on the screen, press CTRL-S. Press CTRL-S again to continue.

- To examine the text in more detail, you can E(dit the program's .TEXT file. On one-drive systems, first use the Filer to T(ransfer the program's .TEXT file from APPLE3: to your boot diskette, APPLE0: or APPLE1:. Then E(dit the file.

If you wish to modify, compile, and execute a new version of a demonstration program, the following diskfiles will be needed:

xxxxxx.TEXT	(any diskette, any drive; required only until read into Editor)
SYSTEM.EDITOR	(any diskette, any drive)
SYSTEM.COMPILER	(any diskette, any drive)
SYSTEM.SYNTAX	(boot diskette, any drive; optional Compiler error messages)
SYSTEM.PASCAL	(boot diskette, boot drive)
SYSTEM.LIBRARY	(boot diskette, boot drive)
SYSTEM.CHARSET	(any diskette, any drive; required if WCHAR or WSTRING used)

One-drive note: Diskette APPLE0: normally contains all the needed files except the demonstration program's .TEXT file. You should use diskette APPLE0: as your boot diskette, and T(ransfer the desired demonstration program's .TEXT file to APPLE0:. Then, with APPLE0: in the disk drive, you are ready to E(dit and R(un the program.

Two-drive note: Using diskette APPLE0: as your boot diskette, put APPLE0: in the boot drive and put APPLE3: in the other drive. You are then ready to E(dit and R(un any program's .TEXT file on APPLE3:.

THE "TREE" PROGRAM

TREE shows the creation of an unbalanced binary tree to sort and retrieve data elements (words, in this case). It lets you specify each new word to be stored in the tree, and then shows you graphically just where the new word was placed in the tree.

When you X(ecute TREE.CODE, you are prompted to

```
ENTER WORD:
```

To quit the program at any time, you can just press the RETURN key in response to this message. To continue, you should type the first word to be sorted (only the first six characters are used). For example, you might type:

```
FLIPPY
```

The program then lists the words entered so far, in alphabetic order.

```
THE WORDS IN ORDER ARE:
FLIPPY
```

No prompting message appears, but you must now press the RETURN key to proceed. When you do, a high-resolution picture is displayed, showing the binary tree as it now exists.

```
BINARY TREE:

          /
         /
|--------|
| FLIPPY |
|--------|
         \
          \
```

The box represents the binary tree's first "node", or sorting element. The node has two "links" which can point the way to further nodes: the upper link in the display can point to nodes which precede this node alphabetically, while the lower link can point to nodes which follow this node alphabetically.

To continue, press the RETURN key again. Again you are prompted to

```
ENTER WORD:
```

Suppose you now type

```
APPLE
```

The program responds

```
THE WORDS IN ORDER ARE:
APPLE
FLIPPY
```

and when you press the RETURN key, another picture of the tree is displayed.

```
BINARY TREE:

                    /
                   /
          |--------|
         /| APPLE  |
        / |--------|
|--------|          \
| FLIPPY |           \
|--------|
         \
          \
```

This is how the word APPLE is placed in the binary tree. The word APPLE is compared to the word in the first node, FLIPPY. Since APPLE precedes FLIPPY, alphabetically, the search continues by following the first node's upper link. If another node is found at the end of that link, APPLE is compared to the word in that node, and the search continues by following that node's appropriate link. The search continues until, on following an appropriate link, no node is found with which to compare APPLE. At that point on the tree, a new node is created, containing APPLE.

Retrieving the words to list them in alphabetic order is harder to describe, although the algorithm is fairly simple.

1. Starting at the root node, FLIPPY, follow the tree taking only the upper link from each node, until a node is found whose upper link does not connect to a further node. The word in this node is the first word, alphabetically, so print it.

2. Now follow this node's lower link.

   a. If a node is connected to the link, follow the tree taking only the upper link from each node, until a node is found whose upper link does not connect to a further node. Print that node's word as the next one in alphabetic order, and repeat step 2.

   b. If no further node is connected to the link, go back down the tree to the node whose upper link led to this node. Print that node's word as the next one in alphabetic order, and repeat step 2. (If no link or a lower link led to this node, the list is complete.)

Remember, to quit this program just press the RETURN key in response to the message

```
ENTER WORD:
```

Caution: You must press the RETURN key two times between each word entry (whether or not you wish to see the tree diagrammed). But if you accidentally press RETURN three times, the program is terminated and your list is lost forever.

Program TREE contains examples of the following:

1. Inserting elements into an unbalanced binary tree (INSERTIT)

2. Retrieving elements in order from such a tree (PRINTTREE)

THE "BALANCED" PROGRAM

BALANCED is identical to TREE, except that it stores words by creating a balanced binary tree. It is taken from an example shown on page 215 of the book "Algorithms + Data Structures = Programs", by Nicklaus Wirth (Prentice-Hall, 1976). An AVL-BALANCED BINARY TREE is rearranged after each element insertion to ensure that, of the two branches at any node, one branch is at most one node longer than the other branch. This method of element insertion is slower than for an unbalanced tree, but subsequent retrieval of elements is faster.

Read the description of the TREE demonstration program for details about using this program. New words are added to the BALANCED tree in the same way described for the unbalanced TREE, but the rearrangement of the BALANCED tree following an insertion is more complex. The words are retrieved in alphabetic order identically in the two programs.

THE "CROSSREF" PROGRAM

CROSSREF is an example of a textual cross-reference generator using an unbalanced binary tree to store and sort words. It is taken from an example shown on page 2Ø6 of the book "Algorithms + Data Structures = Programs", by Nicklaus Wirth (Prentice-Hall, 1976).

When you X(ecute CROSSREF.CODE, you are prompted for the name of an

```
INPUT FILE?
```

Respond by typing the filename of a text file that you wish cross-referenced, on any available diskette. It is not necessary to specify the filename's .TEXT suffix. For example, you might type

```
APPLEØ:MYSTUFF
```

The program then prompts you to specify a

```
DESTINATION FILE?
```

for the resulting cross-referenced list. You should respond by typing

```
CONSOLE:
```

if you want the list to appear on the screen, or

```
PRINTER:
```

if you want the list to be printed on your printer (which must be connected and turned on).

First, the INPUT text file is displayed on the screen or printed, with each line of text numbered. The words of the text are then stored in alphabetic order in a binary tree, one word to each node. A word is defined as beginning with an alphabetic character and containing all subsequent characters until the next non-alphanumeric character. Finally, the text's words are displayed or printed in alphabetic order, each word followed by the text line numbers where that word appears.

Program CROSSREF contains examples of the following:

1. Set membership (TYPE defines items of the tree structure)

2. Sorting into a binary tree

3. Listing from a binary tree (PRINTTREE, also shows recursion)

For more information about tree-sorting, see the demonstration programs TREE and BALANCED.

THE "SPIRODEMO" PROGRAM

SPIRODEMO demonstrates the basic TURTLEGRAPHICS maneuver: move the pen in a straight line, turn, move again in a straight line, turn again, and so on.

The program lets you specify an ANGLE and a CHANGE, and then draws a pattern on the screen. To make the pattern, SPIRODEMO moves the pen one unit, turns through ANGLE, moves 1+CHANGE, turns ANGLE, moves 1+CHANGE+CHANGE, turns ANGLE, etc.

When you X(ecute SPIRODEMO.CODE, this message appears:

```
WELCOME TO WHILEPLOT
ENTER ANGLE Ø TO QUIT.

ANGLE:
```

If you wish to leave the program at any time, just wait until this prompting message is displayed, and then respond by typing a zero and pressing the RETURN key. If you want to continue, type any positive or negative integer to specify the angle (in degrees) through which you wish the TURTLEGRAPHICS pen to turn between each move. For example, you might respond by typing

```
89
```

This tells the pen to turn clockwise, slightly less than a right angle between each move. Now you are asked to specify a

```
CHANGE:
```

Starting with a straight-line pen move of one unit, each subsequent move will increase in length by an amount specified by CHANGE. You must respond by typing a positive integer greater than zero. For example, to make each line one unit longer than the previous line, you would type

```
1
```

When you press the RETURN key, program SPIRODEMO (alias WHILEPLOT) begins to draw its design on the screen, using the parameters that you specified.

On completion of the design, the program continues to display the design until you press any key on the Apple's keyboard. Just press the Apple's spacebar, and the original prompt message will replace the design on the screen. You are then ready to specify a new CHANGE and DISTANCE for

another design (or specify an ANGLE of zero to quit the program).

Caution: This program dies if the first character of an ANGLE or CHANGE response is not a plus sign, a minus sign, or a numeric digit.

Program SPIRODEMO contains examples of the following:

1. Using the TURTLEGRAPHICS unit, including the KEYPRESS function

2. Reading the keyboard buffer without echoing on the screen

THE "HILBERT" PROGRAM

HILBERT shows an historically famous example of recursion, using a space-filling design to create an attractive display on the screen.

You can determine the density of the space-filling design by specifying an integer ORDER from 1 through 7.

When you X(ecute HILBERT.CODE, this message appears:

```
ENTER ORDER Ø TO QUIT.

ORDER:
```

If you wish to quit the program at any time, wait until this message appears, and then type a zero. If you wish to continue, you must type an integer from 1 through 7. An ORDER of 1 fills the space most "loosely", taking barely one repetition of the design to fill the screen. Each higher order fills the screen more and more densely, by repeating the basic design on a smaller and smaller scale. Order 7 fills the screen to solid white, and takes quite a long time doing it. There is no way to stop a display while it is being created, except to press the RESET key. To get the idea, respond by typing

```
4
```

On completion of the design, the program continues to display the design until you press any key on the Apple's keyboard. Just press the Apple's spacebar, and the original prompt message will replace the design on the screen. You are then ready to specify a new ORDER for another design (or specify an ORDER of zero to quit the program).

Caution: This program is terminated if the ORDER response is not a numeric digit from 1 through 7.

THE "GRAFDEMO" PROGRAM

GRAFDEMO is a collection of interesting graphical displays generated by a number of very useful procedures.

The program runs without any interaction; just watch the pretty pictures and then study GRAFDEMO.TEXT to see examples of how these things can be done using TURTLEGRAPHICS. You may even find it handy to use some of GRAFDEMO's procedures directly, in your own programs.

When you X(ecute GRAFDEMO.CODE, this unusual message appears:

```
PRESS ANY KEY TO QUIT.
PLEASE WAIT WHILE CREATING BUTTERFLY
```

Just wait; soon you will see butterflies and many other graphical marvels. Pressing any key on the Apple keyboard will terminate this program on completion of whichever display is currently being created.

Program GRAFDEMO contains examples of the following:

1. Using TURTLEGRAPHICS to draw frames, crosshatching, etc.

2. Creation of an array (BUTTER) for use by procedure DRAWBLOCK

3. Handling of a procedure that is too long, by breaking it into smaller parts (BUTTER) and calling those parts from another procedure (INITBUTTER)

THE "GRAFCHARS" PROGRAM

GRAFCHARS shows the characters found in the file SYSTEM.CHARSET, and their use from TURTLEGRAPHICS. The program runs without interaction.

When you X(ecute GRAFCHARS.CODE, this message appears:

```
PRESS RETURN FOR MORE...
```

From here on, each time you press the Apple's RETURN key another display is placed on the screen. The first display shows all the characters available in SYSTEM.CHARSET . When you have examined any display to your satisfaction, just press the RETURN key again to go on to the next display.

Program GRAFCHARS contains examples of the following:

1. All the upper-case, lower-case, and special characters available through TURTLEGRAPHICS

2. Use of TURTLEGRAPHICS' WCHAR and WSTRING functions

3. How to put a border around a string (BOXSTRING)

4. Use of CHARMODE to keep the characters' boundaries from interfering with the background

THE "DISKIO" PROGRAM

DISKIO shows a sample use of random-access disk files, with terminal-independent output.

Note: This program is NOT a real application, and it is definitely NOT a data-base manager. Its only purpose is to demonstrate some of the principles that would be involved in writing a real file-handling program.

When you X(ecute DISKIO.CODE, you are asked to specify a

```
FILE NAME:
```

You should type a valid disk-file identifier. For example, you might respond by typing

```
APPLEØ:MYFILE.TEXT
```

The program looks on the specified diskette (or the default diskette) for a file with the specified filename. If an existing file by that name is found, it is opened and the main program command prompt line is displayed. If no file by that name is found, the program asks if it should

```
START A NEW FILE?
```

If you type N for No, you will again be asked to type a FILE NAME. There is no exit from the program at this point except by successfully opening a file or by pressing the RESET key. If you type Y for Yes, the program asks

```
RESERVE HOW MANY RECORDS?
```

Respond by typing an integer that specifies the number of records your new file will initially contain. For example, if you type

```
6
```

your new file will start out containing seven records, numbered Ø through 6.

Now the program's main command prompt line appears on the screen:

```
V(IEW  C(HANGE  N(EXT  F(ILE  Q(UIT
```

Typing a V for V(iew causes this message to appear:

```
VIEW WHICH RECORD?
```

You should respond by typing a number from zero through the maximum record number in your file. For instance, typing

```
5
```

lets you view the contents of record number 5.

If you then wish to view the contents of the next record, type N for N(ext. In this way, you can look at as many records as you wish.

Typing a C for C(hange causes this message to appear:

```
CHANGE WHICH RECORD?
```

Again, you should respond by typing a number from zero through the maximum record number in your file. For instance, typing

```
5
```

lets you change the contents of record number 5. To change an entry, just start typing. To leave an entry as it is, and go on to the next entry, just press the RETURN key.

If you then wish to change the contents of the next record, type N for N(ext. In this way, you can change as many records as you wish.

If the N(ext command takes you beyond the last record specified for your file, the program will attempt to extend the file by appending additional records. This is possible if

1. there is room for the record in the current last block of the file, or
2. the next contiguous block on the diskette is available for use by this file.

If it is not possible to extend your file, a message appears to inform you of the problem. You can then type Q to Q(uit this program, enter the Filer, and move files on the diskette until your file has a few free blocks immediately following it. (Use the Filer's E(xtended List command to see the locations of free blocks.) Then you are ready to X(ecute DISKIO again, and extend your file with additional records.

Typing F for F(ile, in response to the main command prompt line, lets you start a new file or reopen another old file. As at the beginning, you are asked for a

```
FILE NAME:
```

Again, there is no exit from this part of the program except to give a successful filename or to press the RESET key.

Program DISKIO contains examples of the following:

1. Terminal-independent output, by reading the file SYSTEM.MISCINFO and using the terminal setup parameters found there (GETCRTINFO)

2. Bullet-proof character input (GETCHAR)

3. Bullet-proof string input, with defaults

4. Use of random-access disk files and system procedure SEEK

5. How to extend a diskette file in place.

APPENDIX B
TABLES

TABLE 1: EXECUTION ERRORS

Ø	System error	FATAL
1	Invalid index, value out of range (XINVNDX)	
2	No segment, bad code file (XNOPROC)	
3	Procedure not present at exit time (XNOEXIT)	
4	Stack overflow (XSTKOVR)	
5	Integer overflow (XINTOVR)	
6	Divide by zero (XDIVZER)	
7	Invalid memory reference <bus timed out> (XBADMEM)	
8	User break (XUBREAK)	
9	System I/O error (XSYIOER)	FATAL
1Ø	User I/O error (XUIOERR)	
11	Unimplemented instruction (XNOTIMP)	
12	Floating point math error (XFPIERR)	
13	String too long (XS2LONG)	
14	Halt, Breakpoint (without debugger in core) (XHLTBPT)	
15	Bad Block	

All FATAL errors require that the system be rebooted. In some cases the system will reboot automatically, and in other cases you will have to reboot it. All other errors cause the system to re-initialize itself.

TABLE 2: I/O ERRORS (IORESULT VALUES)

Ø	No error
1	Diskette has bad Block: parity error (CRC). (Not used on the Apple.)
2	Bad device (volume) Number
3	Bad Mode: illegal operation. (For example, an attempt to read from PRINTER:.)
4	Undefined hardware error. (Not used on the Apple.)
5	Lost device: device is no longer on-line, after successfully starting an operation using that device.
6	Lost file: file is no longer in the diskette directory, after successfully startng an operation using that file.
7	Bad title: illegal file name. (For example, filename is more than 15 characters long.)
8	No room: insufficient space on the specified diskette. (Files must be stored in contiguous diskette blocks.)
9	No device: the specified volume is not on line
1Ø	No file: The specified file is not in the directory of the specified volume.
11	Duplicate file: attempt to rewrite a file when a file of that name already exists.
12	Not closed: attempt to open an open file.
13	Not open, attempt to access a closed file.
14	Bad format, error in reading real or integer. (For example, your program expects an integer input but you typed a letter.)
15	Ring buffer overflow: characters are arriving at the Apple faster than the input buffer can accept them.
16	Write-protect error: the specified diskette is write-protected.
64	Device error: failed to complete a read or write correctly (bad address or data field on diskette).

See Chapter 3 for description of the built-in function IORESULT.

TABLE 3: RESERVED WORDS

These are words that have fixed meanings in Pascal. You can never use them as identifiers without causing a compiler error. The next two tables list some more words you should not use as identifiers.

STANDARD PASCAL RESERVED WORDS

AND
ARRAY
BEGIN
CASE
CONST
DIV
DO
DOWNTO
ELSE
END
FILE
FOR
FORWARD
FUNCTION
GOTO
IF
IN
LABEL
MOD
NIL
NOT
OF
OR
PACKED
PROCEDURE
PROGRAM
RECORD
REPEAT
SET
THEN
TO
TYPE
UNTIL
VAR
WHILE
WITH

ADDITIONAL APPLE PASCAL RESERVED WORDS

EXTERNAL
IMPLEMENTATION
INTERFACE
SEGMENT
UNIT
USES

TABLE 4: PREDEFINED IDENTIFIERS

These are the identifiers of the built-in procedures and functions and the predefined types and variables of Apple Pascal. The list does not include those identifiers that are declared or defined in the special UNITs supplied for the Apple (see next table). If you declare or define one of these identifiers in your program, no error will result but you will lose the capability of the corresponding built-in or predefined entity.

With each identifier, a code is shown in {brackets} to indicate what kind of object the identifier represents. The codes are

{p} PROCEDURE
{b} BOOLEAN FUNCTION
{t} TYPE
{k} CONSTANT
{s} STRING FUNCTION
{i} INTEGER FUNCTION
{r} REAL FUNCTION
{c} CHAR FUNCTION
{f} FILE
{-} OTHER

ABS {r}
BLOCKREAD {i}
BLOCKWRITE {i}
BOOLEAN {t}
CHAR {t}
CHR {c}
CLOSE {p}
CONCAT {s}
COPY {s}
DELETE {p}
EOF {b}
EOLN {b}
EXIT {p}
FALSE {k}
FILLCHAR {p}
GET {p}
GOTOXY {p}
HALT {p}
INPUT {f}
INSERT {p}
INTEGER {t}
INTERACTIVE {t}
IORESULT {i}
KEYBOARD {f}
LENGTH {i}
MARK {p}
MAXINT {k}
MEMAVAIL {i}
MOVELEFT {p}
MOVERIGHT {p}
NEW {p}
ODD {b}
ORD {i}
OUTPUT {f}
PAGE {p}
POS {i}
PRED {-}
PUT {p}
PWROFTEN {r}
READ {p}
READLN {p}
REAL {t}
RELEASE {p}
RESET {p}
REWRITE {p}
ROUND {i}
SCAN {i}
SEEK {p}
SIZEOF {i}
SQR {r}
STR {s}
STRING {t}
SUCC {-}
TEXT {t}
TREESEARCH {i}
TRUE {k}
TRUNC {i}
UNITBUSY {b}
UNITCLEAR {p}
UNITREAD {p}
UNITWAIT {p}
UNITWRITE {p}
WRITE {p}
WRITELN {p}

TABLE 5: IDENTIFIERS DECLARED IN SUPPLIED UNITS

These identifiers are effectively declared or defined only if your program USES their respective UNITs. If your program USES a UNIT and you attempt to declare or define one of the identifiers belonging to that UNIT, you will get a compiler error message 1Ø1: "Identifier declared twice." However if your program doesn't USE a particular UNIT you can make free use of the identifiers of that UNIT.

With each identifier, a code is shown in {brackets} to indicate what kind of object the identifier represents. The codes are

{p} PROCEDURE	{i} INTEGER FUNCTION
{b} BOOLEAN FUNCTION	{r} REAL FUNCTION
{t} TYPE	

TURTLEGRAPHICS UNIT

CHARTYPE {p}	PENCOLOR {p}	TURTLEX {i}
DRAWBLOCK {p}	SCREENBIT {b}	TURTLEY {i}
FILLSCREEN {p}	SCREENCOLOR {t}	VIEWPORT {p}
GRAFMODE {p}	TEXTMODE {p}	WCHAR {p}
INITTURTLE {p}	TURN {p}	WSTRING {p}
MOVE {p}	TURNTO {p}	
MOVETO {p}	TURTLEANG {i}	

APPLESTUFF UNIT

BUTTON {i}	RANDOM {i}
KEYPRESS {b}	RANDOMIZE {p}
NOTE {p}	TTLOUT {p}
PADDLE {i}	

TRANSCEND UNIT

ATAN {r}	LOG {r}
COS {r}	SIN {r}
EXP {r}	SQRT {r}
LN {r}	

TABLE 6: COMPILER ERROR MESSAGES

When the Pascal Compiler discovers an error in your program, it reports that error immediately, by error number. If you then enter the Editor to fix that error, a more complete error message is given, taken from the boot diskette file SYSTEM.SYNTAX . If you remove the file SYSTEM.SYNTAX from the boot diskette, errors will be reported by number, only.

The Pascal Compiler error message corresponding to each error number is given in the table below. Some people will prefer to gain some additional space on their boot diskette, by removing SYSTEM.SYNTAX and using this table instead. You can also print your own copy of this table by T(ransferring the file SYSTEM.SYNTAX to a printer.

```
 1: Error in simple type
 2: Identifier expected
 3: 'PROGRAM' expected
 4: ')' expected
 5: ': ' expected
 6: Illegal symbol (possibly missing ';' on line above)
 7: Error in parameter list
 8: 'OF' expected
 9: '(' expected
1Ø: Error in type
11: '[' expected
12: ']' expected
13: 'END' expected
14: ';' expected (possibly on line above)
15: Integer expected
16: '=' expected
17: 'BEGIN' expected
18: Error in declaration part
19: Error in <field-list>
2Ø: '.' expected
21: '*' expected
22: 'Interface' expected
23: 'Implementation' expected
24: 'Unit' expected

5Ø: Error in constant
51: ': =' expected
52: 'THEN' expected
53: 'UNTIL' expected
54: 'DO' expected
55: 'TO' or 'DOWNTO' expected in for statement
56: 'IF' expected
57: 'FILE' expected
58: Error in <factor> (bad expression)
59: Error in variable

1Ø1: Identifier declared twice
```

102: Low bound exceeds high bound
103: Identifier is not of the appropriate class
104: Undeclared identifier
105: Sign not allowed
106: Number expected
107: Incompatible subrange types
108: File not allowed here
109: Type must not be real
110: <tagfield> type must be scalar or subrange
111: Incompatible with <tagfield> part
112: Index type must not be real
113: Index type must be a scalar or a subrange
114: Base type must not be real
115: Base type must be a scalar or a subrange
116: Error in type of standard procedure parameter
117: Unsatisfied forward reference
118: Forward reference type identifier in variable declaration
119: Re-specified parameters not OK for a forward declared procedure
120: Function result type must be scalar, subrange or pointer
121: File value parameter not allowed
122: A forward declared function's result type can't be re-specified
123: Missing result type in function declaration
124: F-format for reals only
125: Error in type of standard procedure parameter
126: Number of parameters does not agree with declaration
127: Illegal parameter substitution
128: Result type does not agree with declaration
129: Type conflict of operands
130: Expression is not of set type
131: Tests on equality allowed only
132: Strict inclusion not allowed
133: File comparison not allowed
134: Illegal type of operand(s)
135: Type of operand must be boolean
136: Set element type must be scalar or subrange
137: Set element types must be compatible
138: Type of variable is not array
139: Index type is not compatible with the declaration
140: Type of variable is not record
141: Type of variable must be file or pointer
142: Illegal parameter solution
143: Illegal type of loop control variable
144: Illegal type of expression
145: Type conflict
146: Assignment of files not allowed
147: Label type incompatible with selecting expression
148: Subrange bounds must be scalar
149: Index type must be integer
150: Assignment to standard function is not allowed
151: Assignment to formal function is not allowed
152: No such field in this record
153: Type error in read
154: Actual parameter must be a variable
155: Control variable cannot be formal or non-local

156: Multidefined case label
157: Too many cases in case statement
158: No such variant in this record
159: Real or string tagfields not allowed
160: Previous declaration was not forward
161: Again forward declared
162: Parameter size must be constant
163: Missing variant in declaration
164: Substitution of standard proc/func not allowed
165: Multidefined label
166: Multideclared label
167: Undeclared label
168: Undefined label
169: Error in base set
170: Value parameter expected
171: Standard file was re-declared
172: Undeclared external file
174: Pascal function or procedure expected

182: Nested units not allowed
183: External declaration not allowed at this nesting level
184: External declaration not allowed in interface section
185: Segment declaration not allowed in unit
186: Labels not allowed in interface section
187: Attempt to open library unsuccessful
188: Unit not declared in previous 'Uses' declaration
189: 'Uses' not allowed at this nesting level
190: Unit not in library
191: No private files
192: 'Uses' must be in interface section
193: Not enough room for this operation
194: Comment must appear at top of program
195: Unit not importable

201: Error in real number - digit expected
202: String constant must not exceed source line
203: Integer constant exceeds range
204: 8 or 9 in octal number

250: Too many scopes of nested identifiers
251: Too many nested procedures or functions
252: Too many forward references of procedure entries
253: Procedure too long
254: Too many long constants in this procedure
256: Too many external references
257: Too many externals
258: Too many local files
259: Expression too complicated

300: Division by zero
301: No case provided for this value
302: Index expression out of bounds
303: Value to be assigned is out of bounds
304: Element expression out of range

350: No data segment allocated
351: Segment used twice
352: No code segment allocated
353: Non-intrinsic unit called from intrinsic unit
354: Too many segments for the segment dictionary

398: Implementation restriction
399: Implementation restriction
400: Illegal character in text
401: Unexpected end of input
402: Error in writing code file, not enough room
403: Error in reading include file
404: Error in writing list file, not enough room
405: Call not allowed in separate procedure
406: Include file not legal
407: Too many libraries

TABLE 7: ASCII CHARACTER CODES

Code		Char	Code		Char	Code		Char	Code		Char
Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex	
0	00	NUL	32	20	SP	64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	89	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

APPENDIX C

ADDITIONAL DETAILS OF TEXT I/O

Here are some facts about READ and READLN that you need to know if you do not follow the suggestions in the "lntroduction to Text I/O" section of Chapter 3. In particular, these facts are important if you mix reading and writing operations on the same diskette textfile. You may also need to know exactly when EOLN and EOF become true with READLN and with numeric variables.

Note that for mixed reading and writing, the rules given below are more straightforward for INTERACTIVE file than for TEXT files.

After READ with a CHAR variable and an INTERACTIVE file:

- The file buffer variable contains the character that was READ, unless EOLN or EOF is true.
- If the next I/O operation is a PUT, WRITE, or WRlTELN, it affects the character after the one that was READ.
- EOF is true if the character READ was the end-of-file character. In this case the value of the file buffer variable is undefined.
- EOLN is true if the character READ was the end-of-line character. In this case the file buffer variable contains a space.
- EOLN is also true if EOF is true.

After READ with a CHAR variable and a TEXT file:

- The file buffer variable contains the character after the character that was READ, unless EOLN or EOF is true.
- If the next I/O operation is a PUT, WRITE, or WRITELN, it affects the second character after the one that was READ.
- EOF is true if the character READ was the last character in the file (not counting the end-of-file character). In this case the value of the file buffer variable is undefined.
- EOLN is true if the character READ was the last character on the line (not counting the end-of-line character). ln this case the file buffer variable contains a space.
- EOLN is also true if EOF is true.

After READ with a numeric variable and a TEXT or INTERACTIVE file:

- The file buffer variable contains the character after the last character of the numeric string that was READ, unless EOLN or EOF is true.
- If the next I/O operation is a PUT, WRITE, or WRITELN, it affects the second character after the last character of the numeric string.
- EOF is true if the last character of the numeric string was the last character in the file (not counting the end-of-file character). In this case the value of the file buffer variable is undefined.
- EOLN is true if the last character of the numeric string was the last character on the line (not counting the end-of-line character). In this case the file buffer variable contains a space.
- EOLN is also true if EOF is true.

After READ with a STRING variable and a TEXT or lNTERACTIVE file:

- The file buffer variable contains a space which represents the end-of-line character at the end of the line, unless EOF is true.
- If the next I/O operation is a PUT, WRlTE, or WRITELN, it affects the first character on the next line.
- EOF is true if the line READ was the last line in the file. In this case the value of the file buffer variable is undefined.
- EOLN is always true.

After READLN with any variable and an lNTERACTIVE file

- The file buffer variable contains a space which represents the end-of-line character at the end of the line, unless EOF is true.
- If the next I/O operation is a PUT, WRITE, or WRlTELN, it affects the first character on the next line.
- EOF is true if the line READ was the last line in the file. In this case the value of the file buffer variable is undefined.
- EOLN is never true.

After READLN with any variable and a TEXT file

- The file buffer variable contains the first character on the next line, unless EOLN or EOF is true.
- lf the next I/O operation is a PUT, WRITE, or WRITELN, it affects the second character on the next line.

- EOF is true if the line READ was the last line in the file. In this case the value of the file buffer variable is undefined.

- EOLN is true only when EOF is true.

APPENDIX D

ONE-DRIVE STARTUP

This appendix is a tutorial session to get you started using the Language System with Pascal, on an Apple II with one diskette drive. If your system has two or more diskette drives, please skip this appendix and read Appendix E instead.

EQUIPMENT YOU WILL NEED

You should have the following:

1. Your 48K Apple computer, with a Language Card installed, and one disk drive attached to the connector marked "DRIVE 1" on the disk controller card. The disk controller card must have the new PROMs, P5A and P6A (which came with the Language System), and must be installed in the Apple's peripheral device slot 6.

2. A TV set or video monitor properly connected to your Apple.

3. The following Language System diskettes:

    a. APPLEØ:
    b. APPLE1:
    c. APPLE2:
    d. APPLE3:
    e. A blank diskette
    f. Another blank diskette

The diskettes marked "APPLE1:" and "APPLEØ:" are needed to start the system. The diskette marked "APPLE2:" adds some extra features to the system (the Assembler and the Linker). You will not need the diskette marked "APPLE2:" until later (many users of single-drive systems will never need it). The diskette marked "APPLE3:" contains a number of useful utility programs, and some interesting demonstrations; Appendix A of this manual explains these demonstrations.

Your Apple and its TV or monitor should be plugged in. Turn on the TV now, so that it can warm up; but leave the Apple turned off.

THE TWO-STEP STARTUP

There are two steps to starting Apple Pascal running on your system.

STEP ONE OF STARTUP

First insert the diskette marked APPLE1: in the disk drive. If you are not familiar with handling diskettes, see the manuals that came with your disk drives. Diskettes must be treated correctly if they are to last.

Close the door to the disk drive, and turn on the Apple. The rest of Step One is automatic. First, the message

```
APPLE II
```

appears at the top of your TV or monitor screen, and the disk drive's "IN USE" light comes on. The disk drive emits a whirring, zickking sound that is as pleasant as a cat's purring, since it lets you know that everything is working. The screen lights up for an instant with a display of black at-signs (@) on a white background, then goes black again. Next, the disk drive stops entirely for a moment; then it whirrs some more. Finally, the message

```
WELCOME  APPLE1,  TO
U.C.S.D.  PASCAL SYSTEM II.1
CURRENT DATE IS 26-JUL-79
```

appears (the date will be different), followed in a second or so by a line at the top of the screen:

```
COMMAND: E(DIT, R(UN, F(ILE, C(OMP, L(IN
```

This line at the top of the screen is called a "prompt line". When you see this prompt line, you know that your Apple computer is running the Apple Pascal system.

If you just wish to edit text and programs, or if you wish to run previously compiled programs, you may stop now. At this point, your system can do most of the things you will normally want to do in Apple Pascal, except for compiling new programs that you write.

However, if you also wish to compile programs that you write, in order to run them, you should proceed to Step Two of the startup procedure.

STEP TWO OF STARTUP

Remove the diskette marked APPLE1: from the disk drive, and insert the one marked APPLEØ: . Close the door to the drive and press the key marked RESET , in the upper right corner of the Apple's keyboard.

The at-signs come back for an instant, and the disk drive whirrs and completely stops for a second, then whirrs some more. The whole process takes about 16 seconds. Finally, the message

```
WELCOME  APPLEØ,  TO
U.C.S.D.  PASCAL SYSTEM II.1
CURRENT DATE IS 26-JUL-79
```

appears (the date will be different), followed in a second or so by the prompt line at the top of the screen

```
COMMAND: E(DIT, R(UN, F(ILE, C(OMP, L(IN
```

Again, this prompt line lets you know that your Apple computer is running the Apple Pascal System.

After completing Step Two of the startup procedure, your system can do all the things you will normally want to do in Apple Pascal: filing, editing, running....and compiling. However, diskette APPLEØ: is missing one file that is needed for the initial startup when you first turn the Apple's power on. That is why you must go through Step One of the startup procedure before going on to Step Two.

CHANGING THE DATE

The date that comes on the diskette will not be correct. It is a good habit to reset the date the first time you use the Apple Pascal System on any given day. It only takes a few seconds. Press F on the keyboard (without pressing the RETURN key or any other keys). The screen goes blank, and then this line appears at the top:

```
FILER: G, S, N, L, R, C, T, D, Q
```

This is a new prompt line. Prompt lines are named after their first word. The prompt line you first saw was the "COMMAND" prompt line. This one is the "FILER" prompt line. Sometimes we say that you are "in the Filer" when this line is at the top of the screen. Each of the letters on the prompt line represents a task that you can ask the system to do. For example, to change the date, press D (again, just type the single key, without pressing RETURN or any other key).

When you do, another message is put on the screen. It says:

```
DATE SET: <1..31>-<JAN..DEC>-<ØØ..99>
TODAY IS 26-JUL-79
NEW DATE ?
```

It doesn't really mean that today is 26-JUL-79 (or whatever date your screen shows), but that the Apple THINKS that is today's date. Since it isn't, you can change the date to be correct. The correct form for typing the date is shown on the second line of the message: one or two digits giving the day of the month, followed by a minus sign, followed by the first three letters of the name of the month, followed by another minus sign, followed by the last two digits of the current year. Then press the key marked RETURN .

If the month and year are correct (as they will often be, when you change the date) all you have to do is type the correct day of the month, and press the RETURN key. The system will assume that you mean to keep the same month and year displayed by the message. If you type a day and a month, the system will assume you mean to keep only the year the same.

Go ahead and make the date correct. This is your first interaction with the system, and is typical of how the system is used. In general, at the top of the screen there will usually be a prompt line which represents several choices of action. When you type the first letter of one of the choices, either you will be shown a new prompt line giving a further list of choices, or else the system will carry out the desired action directly. If you type a letter that does not correspond to one of the choices, the prompt line blinks but otherwise nothing happens. Remember to type only a single letter to indicate your choice; it is not necessary to press the RETURN key afterward.

Sometimes, as when setting the date, you are asked to type a response of several characters. You tell the system that your response is complete by pressing the RETURN key. If you make a typing error before pressing the RETURN key, you can back up and correct the error by pressing the left-arrow key. You should experiment by making deliberate errors in entering a date, and then erasing the errors with the left-arrow key.

One further note. Normally, your new date is saved on the diskette, so the system "remembers" this date the next time you turn the Apple on. However, since you are using the write-protected diskettes that came with your Language System, your new date was not permanently saved. The next time you turn the Apple off, the new date will be "forgotten". By the end of this session, you will have made backup copies of the Language System diskettes. From then on, you will use these copies, which are not write-protected, and your date changes will be saved correctly.

MAKING BACKUP DISKETTE COPIES

WHY WE MAKE BACKUPS

Ask yourself this question: What would happen to your system if you were to lose or damage one of the system diskettes (APPLEØ:, APPLE1:, APPLE2:, or APPLE3:)? It would be as bad as losing your Apple, as far as your being able to use Pascal.

These diskettes are quite precious. The first thing you should do, therefore, is to make backup copies of them. Afterward, you should never use the originals, but put them someplace where the temperature is moderate, where there is no danger of them getting wet, and where such diskette destroyers as dogs, dirt, children, and magnetic fields cannot get at them.

A truly cautious person will keep on hand two backup copies of each original. That way, you will need to use an original only in the very rare case when both of its backup copies are lost (when one copy is lost or damaged, another backup copy is made from the surviving backup copy). If your backups were damaged or erased while in use, find out why they were destroyed before inserting your only surviving copy. Using diskettes for which you have backups, repeat the procedure that destroyed the first diskettes; if you can't figure out what the problem

is, take your system to the dealer to make sure it is working correctly.

HOW WE MAKE BACKUPS

The Apple Pascal system can copy all the information from one diskette (or any portion of the information) onto another diskette. But the system cannot store information on a new diskette, just as that diskette comes from the computer store. Therefore, the system is supplied with a program that allows you to take any 5-inch floppy diskette and "format" it so that it will work with the Apple Pascal system.

Incidentally, this is one of the nice little things about the Apple system: ANY high-quality 5-inch floppy diskette (Apple recommends diskettes made by Dysan Corporation) will work on it. Some systems require you to have "1Ø sector" or "15 sector"or "soft sectored" diskettes. The Apple doesn't care, it takes any of these kinds of diskettes, and (through the FORMATTER program) makes them into the kind of diskette it needs.

If you have been following this discussion by carrying out the instructions on your Apple, the FILER prompt line should be showing at the top of the screen:

```
FILER: G, S, N, L, R, C, T, D, Q
```

Type Q on the keyboard to Quit the Filer.

GETTING THE BIG PICTURE

When you Quit the Filer, the disk whirrs, and you see the COMMAND prompt line again:

```
COMMAND: E(DIT, R(UN, F(ILE, C(OMP, L(IN
```

There is actually more of this prompt line, off to the right of your TV or monitor. To see the rest of the screen, hold down the key marked CTRL and, while holding it down, press the A key right alongside it. (Or, to be brief, we say: "press CTRL-A".)

You now see

```
K, X(ECUTE, A(SSEM, D(EBUG,?
```

This is simply the rest of the line that began "COMMAND:". All together, the full prompt line would look like this:

```
COMMAND: E(DIT, R(UN, F(ILE, C(OMP, L(INK, X(ECUTE, A(SSEM, D(EBUG,?
```


The Apple Pascal system displays information on a "screen" that is 8Ø characters wide, but your TV or monitor shows only the leftmost 4Ø characters or the rightmost 4Ø characters at any one time. You use the CTRL-A trick whenever you wish to see if there is more stuff on the other "half" of the screen. Repeated pressing of CTRL-A flips back and forth between the left half of the screen and the right half. Also, sometimes the TV display will seem to be blank. This might mean that you are just staring at the empty right half of the screen. Before you come to the conclusion that something is wrong, always try CTRL-A. You get back to the left side of the screen by typing CTRL-A again, and you might find that everything is OK after all.

Summary of this digression: The screen is really twice as wide as it looks. To flip from the left side to the right side or back again, you type CTRL-A.

FORMATTING NEW DISKETTES

When the COMMAND prompt line is showing at the top of the screen, remove your system diskette (APPLE1: or APPLEØ:) from the disk drive and place the diskette APPLE3: in the drive. This has to be done because the FORMATTER program is on APPLE3: . Now, type

```
X
```

and the screen responds:

```
EXECUTE WHAT FILE?
```

You type

```
APPLE3: FORMATTER
```

and press the RETURN key. The disk whirrs a bit and the screen says:

```
APPLE DISK FORMATTER PROGRAM
FORMAT WHICH DISK (4, 5, 9..12) ?
```

Now comes a grand session. Take all the new, blank diskettes that you are going to use with the Apple Pascal System (but not, of course, any diskettes that have precious information on them, such as the diskettes that came with the Apple Pascal System) and place them in a pile. Their labels should be blank. Make sure that you don't have any diskettes with data in a non-Pascal format, such as BASIC diskettes: the Apple Pascal system will be unable to read them, and will regard them as blank, erasing any old information in the formatting process.

Remove the diskette APPLE3: from the disk drive, and place one of the blank diskettes into the drive. Type

```
4
```

and press the RETURN key.

If the diskette in the drive has already been formatted, you will receive a warning. For example, if you have left APPLE3: in the drive you will be warned with the message

```
DESTROY DIRECTORY OF APPLE3 ?
```

At this point you can type

```
N
```

(which stands for "No") without pressing the RETURN key, and your diskette will not be destroyed.

Let's assume that you have placed a new, unformatted diskette in the disk drive. Then you will not get any warning, but the Apple will place this message on the screen:

```
NOW FORMATTING DISKETTE IN DRIVE 4
```

The drive will make some clickings and buzzings and begin to whirr and zick. The process takes about 32 seconds. When formatting is complete, the screen again shows the message

```
FORMAT WHICH DISK (4, 5, 9..12) ?
```

Now you have a formatted diskette. We suggest that you write the word "Pascal" in small letters at the top of the diskette's label, using a marking pen. Do not use a pencil or ballpoint pen, as the pressure may damage the diskette. The label will let you know that the diskette is formatted for use with the Apple Pascal system, and you can distinguish it from unformatted diskettes, BASIC diskettes, or diskettes for use with other systems.

While you are at it, repeat this formatting process on all the new diskettes that you want to use with the Apple Pascal System. With each new diskette, place it in the disk drive, type 4 and press the RETURN key.

You may wonder why your one-and-only disk drive is called "4". There's no good reason for this, it's just that the disk drive was assigned the number 4. Why, in Spanish, is the word for window "ventana"? It just happened that way.

When you have finished formatting all your new diskettes, and have written the word "Pascal" on each of them, answer the question

```
FORMAT WHICH DISK (4, 5, 9..12) ?
```

with a simple press of the key marked RETURN . You get the message

```
PUT SYSTEM DISK IN #4 AND PRESS RETURN
```

By "SYSTEM DISK" the Apple means "APPLEØ:" (unless you stopped after Step One of the startup procedure, and continued to use APPLE1: as your system disk). By "#4" the Apple means the disk drive. Sometimes your disk drive is called "DRIVE 4" and sometimes "#4:", but it's all the same thing.

Do as it says, place the diskette marked APPLEØ: in the disk drive (or, as we say in Apple Pascal jargon, "Put APPLEØ: in #4:") and press the RETURN key.

The Apple says:

```
THAT'S ALL FOLKS...
```

And if you watch the top of the screen, the line:

```
COMMAND: E(DIT, R(UN, F(ILE, C(OMP, L(INK, X(ECUTE, A(SSEM, D(EBUG,?
```

appears (of course, it doesn't all appear; but you know it's there, and can check with CTRL-A).

MAKING THE ACTUAL COPIES

As you have seen, you can get into the Filer by typing F when you have the COMMAND prompt line on the screen. You must have diskette APPLE1: or diskette APPLEØ: in the disk drive when you type F for the Filer, or (if APPLEØ: is your system diskette) you will get the message

```
NO FILE APPLEØ:SYSTEM.FILER
```

If this happens, just put APPLEØ: in the disk drive and type F again.

The Filer is that portion of the system that allows you to manipulate information on diskettes. One of the Filer's abilities is to transfer information from one diskette to another. To invoke this facility, once you have the FILER prompt line on the screen, type T for T(ransfer.

This is what you see:

```
TRANSFER ?
```

Place diskette APPLE3: into the disk drive and answer the question as follows:

```
APPLE3:
```

which means that you want to transfer the entire contents of the source diskette called APPLE3: . After you have specified which diskette's information you want transferred (and pressed the key marked RETURN), the computer checks to make sure the correct diskette is in the disk drive. If you have forgotten to put diskette APPLE3: in the drive, then you will see the message

```
APPLE3:
NO SUCH VOL ON-LINE <SOURCE>
```

In that case you must type T for Transfer again, and repeat the process. With the correct source diskette in the drive, the Transfer process continues and the computer asks the next obvious question: If you are going to transfer something, then

```
TO WHERE ?
```

Answer this question by typing

```
BLANK:
```

This is the name of the destination diskette, onto which you want APPLE3:'s information transferred. "BLANK:" is any of the diskettes that you just formatted. When a diskette is formatted it is automatically given the name BLANK: . Incidentally, those colons (:) are very important. You use them to indicate that you are referring to an entire diskette, and not just a part of one.

After you have told the computer where you want APPLE3:'s information transferred (and pressed the key marked RETURN), it says:

```
TRANSFER 28Ø BLOCKS ? (Y/N)
```

This message is mainly there to give you a chance to abandon the transfer if you made a typing error in the names of the source or the destination diskettes. The phrase "28Ø BLOCKS" means merely "THE WHOLE DISKETTE". In any case, you type

```
Y
```

The disk whirrs and zicks a few times, and you see the message:

```
PUT IN BLANK:
TYPE <SPACE> TO CONTINUE
```


Do as it says. By the colon, you know that it means to put the diskette called BLANK: into the disk drive. The second line tells you to press the space bar when the diskette is in place (and the door closed, of course).

All the information which is on diskette APPLE3:, including the diskette's name, will be copied onto diskette BLANK:, completely overwriting BLANK:. Therefore, the computer warns you that you are about to lose any information that might be stored on BLANK:. It says

```
DESTROY BLANK: ?
```

Since you want to turn BLANK: into a perfect copy of APPLE3:, the answer is

```
Y
```

The process is under way. The computer will tell you to first put in one diskette and then the other. Follow the instructions. Your screen will look like this after a while:

```
PUT APPLE3: IN UNIT #4
TYPE <SPACE> TO CONTINUE
PUT BLANK: IN UNIT #4
TYPE <SPACE> TO CONTINUE
PUT APPLE3: IN UNIT #4
TYPE <SPACE> TO CONTINUE
PUT BLANK: IN UNIT #4
TYPE <SPACE> TO CONTINUE
PUT APPLE3: IN UNIT #4
TYPE <SPACE> TO CONTINUE
PUT BLANK: IN UNIT #4
TYPE <SPACE> TO CONTINUE
PUT APPLE3: IN UNIT #4
TYPE <SPACE> TO CONTINUE
PUT BLANK: IN UNIT #4
TYPE <SPACE> TO CONTINUE
```

and so on. You will have to insert the two diskettes a total of 2Ø times, and press the spacebar 2Ø times, to copy the entire diskette. When copying is done, the screen celebrates by saying

```
APPLE3:                    --> BLANK:
```

By this cryptic remark, the computer is telling you that the contents of APPLE3: , including the diskette's name, have been copied onto the diskette that used to be called BLANK: . This is just what you wanted. Now, writing lightly with a marking pen (do not use a pencil or a ballpoint pen), write "APPLE3:" on the new diskette's label. It is very important to label diskettes immediately, so you know what information they contain.

DO IT AGAIN, SAM

You should, at this time, make sure that you have at least one backup copy of each of the Pascal system diskettes: APPLEØ:, APPLE1:, APPLE2:, and APPLE3: . Then you should store the original diskettes away in a safe place.

When you are through making backup copies, be sure to put APPLEØ: (or APPLE1: if you are using that as your system diskette) back into the disk drive, BEFORE typing Q to Quit the Filer. If you forget to do this, the system will stop responding to the keyboard after you type Q ; you will have to turn the Apple off and repeat the entire startup procedure.

USING THE SYSTEM

A DEMONSTRATION

At last, the reward for all your work to this point: you are finally ready to use the Apple Pascal system to run a program. Diskette APPLE3: contains several small "demonstration" programs. To see a list of those programs, put APPLEØ: in the disk drive and enter the Filer (by typing F in response to the COMMAND prompt line, remember?). When the FILER prompt line appears on the screen, put APPLE3: in the drive and type L to List the diskette's directory. The Filer says:

```
DIR LISTING OF ?
```

In response, type the name of the diskette whose directory you wish to see:

```
APPLE3:
```

When you press the RETURN key, a long list of program files appears on the screen, many of them both in their .TEXT versions (the form in which they are written and edited) and also in their compiled .CODE versions (the form in which they can be executed). When the screen is full, the display stops and the message

```
TYPE <SPACE> TO CONTINUE
```

appears at the top of the screen. Press the Apple's spacebar to see the remaining files. For now, we are interested in the file named GRAFDEMO.CODE . But before executing this program, you must Transfer it to your system diskette, APPLEØ: (most graphics programs must use routines from the "system library", a file on APPLEØ: and also on APPLE1:). In response to the FILER prompt line, type

```
T
```


The Filer says

```
TRANSFER ?
```

Answer the question as follows:

```
APPLE3:GRAFDEMO.CODE
```

which means you want to transfer only the file named GRAFDEMO.CODE from the source diskette named APPLE3: . The Filer checks to see that APPLE3: is in the disk drive, and that it contains a file named GRAFDEMO.CODE, and then asks

```
TO WHERE ?
```

You know that you want a copy of the file GRAFDEMO.CODE transferred to the destination diskette APPLEØ: . To avoid confusion, let's give this copied file the same name when it is transferred to APPLEØ: . To do this, answer the question by typing

```
APPLEØ:GRAFDEMO.CODE
```

Note: you MUST specify a name for the file on the destination diskette. If you forget to type a file name, the Filer thinks you are referring to the entire diskette, and asks

```
DESTROY APPLEØ: ?
```

Since you do not wish to destroy APPLEØ: , type

```
N
```

Now, if you have typed all of your responses correctly, a new display appears:

```
PUT IN APPLEØ:
TYPE <SPACE> TO CONTINUE
```

Follow the directions, putting APPLEØ: in the disk drive and pressing the Apple's spacebar. You are soon rewarded with the message

```
APPLE3:GRAFDEMO.CODE
 --> APPLEØ:GRAFDEMO.CODE
```

This tells you that a copy of the file GRAFDEMO.CODE on diskette APPLE3: has been successfully transferred to a file named GRAFDEMO.CODE on diskette APPLEØ: . Since the system diskette APPLEØ: is already in the disk drive, you may now safely type Q to Quit the Filer. When the COMMAND prompt line appears, type X for X(ecute. The Apple says

```
EXECUTE WHAT FILE?
```

Answer by typing the name of the file you just transferred to APPLEØ:

```
APPLEØ:GRAFDEMO
```

Note: DO NOT type the suffix .CODE ; the system knows you can execute only a code file, so it automatically supplies the suffix .CODE for you, in addition to any name that you type.

When this message appears:

```
PRESS ANY KEY TO QUIT.
PLEASE WAIT WHILE CREATING BUTTERFLY
```

the program is running. After a short pause, the display begins. Just sit back and enjoy it: soon you'll be writing your own programs yourself. When you are tired of watching, press the spacebar on the Apple's keyboard to return to the COMMAND prompt line. You can use this same procedure to run any of the programs on APPLE3: . These programs and their purposes are described in the Appendix A.

DO IT YOURSELF

Now, for some more experience at using the Apple Pascal system, let's try writing a little program. This discussion will assume that you are using your new copy of APPLEØ: as your system diskette (or "boot diskette" as it is often called). This copy is not write-protected and you have never used the Editor to create any new files on it before (it's all right if you have added the file GRAFDEMO.CODE to it).

With the COMMAND prompt line showing, and with APPLEØ: in the disk drive, type E to select the E(dit option. Soon, this message appears:

```
>EDIT:
NO WORKFILE IS PRESENT. FILE? ( <RET> FOR NO FILE <ESC-RET> TO EXIT )
:
```

As usual, you must use CTRL-A to see the right half of the message. This message gives you some information and some choices. The first word, >EDIT: , tells you that you are now in the Editor. The next sentence, NO WORKFILE IS PRESENT , tells you that you have not yet used the Editor to create a "workfile", which is a "scratchpad" diskette copy of a program you are working on. If there had been a workfile on APPLEØ: , that file would have been read into the Editor automatically.

Since there was no workfile to read in, the Editor asks you, FILE? If you now typed the name of a .TEXT file stored on APPLEØ:, that textfile would be read into the Editor. However, there are no .TEXT files on APPLEØ: yet, and besides, you want to write a new program. In parentheses, you are shown how to say that you don't want to read in an old file: <RET> FOR NO FILE . This means that, if you press the Apple's RETURN key, no file will be read in and you can start a new file of your own. That's just what you want to do, so press the Apple's RETURN key

(the rest of the message says if you first press the ESC key and THEN press the RETURN key, you'll be sent back to the COMMAND prompt line). When you have pressed only the RETURN key, the full EDIT prompt line appears:

```
>EDIT: A(DJST C(PY D(LETE F(IND I(NSRT J(MP R(PLACE Q(UIT X(CHNG Z(AP
```

The chapter called THE EDITOR in the Apple Pascal Operating System Manual explains all of these command options in detail; for now you will only need a few of them. The first one you will use is I(NSRT , which selects the Editor's mode for inserting new text. Type I to select Insert mode, and this prompt line appears:

```
>INSERT: TEXT [<BS> A CHAR,<DEL> A LINE] [<ETX> ACCEPTS,<ESC> ESCAPES]
```

As long as this line is showing at the top of the screen, anything you type will be placed on the screen, just to the left of the white square "cursor". If the cursor is in the middle of a line, the rest of the line is pushed over to make room for the new text. If you make a mistake, just use the left-arrow key to backspace over the error, and then retype. At any time during an insertion, if you press the Apple's ESC key your insertion will be erased. At any time during an insertion, if you press CTRL-C the insertion will be made a permanent part of your file, safe from being erased by ESC or by the left-arrow key. You can then type I to reenter Insert mode and type more text.

Now for our program. With the INSERT prompt line showing, press the RETURN key a couple of times, to move the cursor down, and then type

```
PRORAFM DEMO;
```

You can use any name for your program, but in this discussion it will be called DEMO . Now press CTRL-C (type C while holding down the CTRL key). Your insertion so far is made "permanent", and the EDIT prompt line reappears. But, horrors! You made several typing errors when typing the word PROGRAM . Since you have already pressed CTRL-C , it is too late to backspace over your errors and retype them.

Fortunately, there are other ways. First, let's correct the missing G in PROGRAM . Using the left arrow key, move the cursor left until it is sitting directly on the R . Then type I to reenter Insert mode. Ignore the fact that the remainder of the line seems to have suddenly disappeared, and type the missing letter G . When you press CTRL-C to make this insertion permanent, the rest of the line returns:

```
PROGRAFM DEMO;
```

The letter F is certainly not needed, so move the cursor right (using the right-arrow key) until it is sitting directly on the F . Now type D to select the Editor's D(LETE option. When the DELETE prompt line appears, press the right-arrow key once. The offending F instantly disappears. What happens next is similar to Insert mode: if you press the ESC key, the deletion is forgotten, as if it had never happened. If you press CTRL-C , the deletion is made a permanent part of your

file. To remove that F permanently, press CTRL-C . The line closes in to fill the deleted letter's place:

```
PROGRAM DEMO;
```

Now you know how to use the Editor's Insert and Delete modes to write text and to correct your errors. Try typing the rest of program DEMO into your file. Be sure to "accept" your insertions, from time to time, by pressing CTRL-C . That way, you minimize your loss if you accidentally press the ESC key. Here is the complete program:

```
PROGRAM DEMO;

USES TURTLEGRAPHICS, APPLESTUFF;
VAR ANGLE, DISTANCE : INTEGER;

PROCEDURE CRAWL;
BEGIN
  MOVE (2 * DISTANCE);
  TURN (ANGLE)
END;

BEGIN
  ANGLE := Ø;
  REPEAT
    INITTURTLE;
    PENCOLOR (WHITE);
    FOR DISTANCE := 1 TO 99 DO CRAWL;
    ANGLE := ANGLE + 5
  UNTIL KEYPRESS;
  TEXTMODE
END.
```

When you are typing this program, the punctuation and spelling must be exactly as shown. The indentation of the lines is not important, but it easier to read as shown. You will notice that, once you have started a new indentation, the Editor maintains that indentation for you. To move back to the left, just press the left-arrow key before you type anything on the new line.

Program DEMO makes use of graphics routines in the Unit TURTLEGRAPHICS, and uses the keypress function from the Unit APPLESTUFF (see Chapter 7 for more details). The third line of the program declares two integer variables, DISTANCE and ANGLE. Next, a Pascal procedure named CRAWL is defined, between the first BEGIN and END; . From here on, each time this new Pascal statement CRAWL is used, a graphics "turtle" will trace a line on the screen, of length 2*DISTANCE moving in the current direction, and will then change the direction by an amount ANGLE.

The next BEGIN and the last END. outline the main program. The portion of the program from REPEAT to UNTIL KEYPRESS is repeated over and over again, until any key on the Apple's keyboard is pressed.

In each repetition, the screen is cleared and the tracing color is set to WHITE. Then the procedure CRAWL is performed, first with the value of DISTANCE set to one, then with DISTANCE set to the value two, and so on, until DISTANCE is set to 99 . The "turtle" moves, then turns, then moves some more, then turns again, and so on, for 99 steps. That completes one design on the screen. In the next repetition, if no key has been pressed, the ANGLE has increased by 5 degrees, the screen is cleared by INITTURTLE, and the whole process starts again.

Now you should save this program. With the EDIT prompt line showing, type Q to select the Q(UIT option. The following message appears:

```
>QUIT:
     U(PDATE THE WORKFILE AND LEAVE
     E(XIT WITHOUT UPDATING
     R(ETURN TO THE EDITOR WITHOUT UPDATING
     W(RITE TO A FILE NAME AND RETURN
```

Type U to create a "workfile" diskette copy of your program (future versions of this file will be "Updates"). This workfile is a file on your boot diskette called SYSTEM.WRK.TEXT . The Apple says

```
WRITING...
YOUR FILE IS 33Ø BYTES LONG.
```

(the number of bytes may be a little different) and then the COMMAND prompt line reappears. Now type R to select the R(UN option. This automatically calls the Compiler for you, since the workfile contains text. If you have typed the program perfectly, the following messages (again, perhaps with slightly different numbers) appear, one by one:

```
COMPILING...


PASCAL COMPILER II.1 [B2B]
<   Ø>....
TURTLEGR [ 2483 WORDS]
<   5>.........................
APPLESTU [ 1Ø78 WORDS]
<  3Ø>...................
CRAWL    [ 1Ø98 WORDS]
<  46>.....
DEMO     [ 11Ø9 WORDS]
<  51>........
59 LINES
SMALLEST AVAILABLE SPACE = 1Ø98 WORDS
```

If the Compiler discovers mistakes, it will give you a message such as

```
PROFRAM <<<<
LINE 2, ERROR 18: <SP>(CONTINUE), <ESC>(TERMINATE), E(DIT
```

Don't despair; just type E for E(DIT . Your workfile will be automatically read back into the Editor for repairs. Read the error

message at the top of the screen, press the spacebar, and make any necessary changes using I(nsert and D(elete. Then Q(uit, U(pdate the workfile, and R(un your program again, by typing Q U R (the Apple will store up several commands in advance).

When your program has been successfully Compiled, it is automatically executed. You will see the message

RUNNING...

and then a horizontal line appears on the screen. That is the first design your program draws: the white "turtle" moves out a distance 2*1 , turns an angle Ø ; moves 2*2 , turns Ø ; moves 2*3 , turns Ø ; etc. Keep watching as successive designs turn through larger and larger angles between moves. When you want to interrupt the program, press any key on the keyboard.

Try making changes to the program, by setting a different starting ANGLE, or a different increment to the ANGLE, or a different distance to MOVE. To do this, type E for E(DIT, use I(NSRT and D(LETE to make changes, and then Q(uit, U(pdate the workfile, and R(un again by typing Q U R . This cycle of Edit-Run-Edit-Run is the basis of all program development in the Apple Pascal system.

The workfile on APPLEØ: now contains the text version of your program in a file named SYSTEM.WRK.TEXT , and the compiled P-code version of your program in another file named SYSTEM.WRK.CODE . When your program is running as you want it to, you should save the text and code workfile under other filenames. With the COMMAND prompt line showing, type F to enter the Filer. When the FILER prompt line appears, type S for S(ave. You will be asked

SAVE AS ?

and you should respond by typing any filename with fewer than 1Ø characters. For example, you might type

DEMO

This changes the names of the workfile from SYSTEM.WRK.TEXT to DEMO.TEXT , and from SYSTEM.WRK.CODE to DEMO.CODE . If you want to keep a permanent copy of your program on another diskette, you should now use the T(ransfer command to transfer DEMO.TEXT and DEMO.CODE, one at a time, to the other diskette. Remember to wait for the prompt message before removing the source diskette from the drive and putting in the destination diskette.

WHAT TO LEAVE IN THE DRIVE

When you turn the Apple off, it is a good idea to leave the diskette called APPLE1: in the disk drive. If some other diskette or no diskette is in the drive when the Apple is turned on, the drive will spin indefinitely. If this continues for hours and hours, some wear will take place on the drive and any diskette in it. So, it is a good idea to make a habit of leaving a copy of APPLE1: (now that you have copies) in the disk drive when you turn the system off. (APPLEØ: will not do, as it is missing a file that is needed for the first stage of system startup.)

Of course, if you turn on the system and APPLE1: is not in the drive, just press the key marked RESET . Place APPLE1: in the drive and turn the system off and then on again. No damage results from turning on the Apple with the wrong diskette (or no diskette) in the drive. Gradual, unnecessary wear results from leaving the disk drive running for a long period of time with the incorrect diskette (or no diskette) in the drive.

ONE-DRIVE SUMMARY

STARTING UP THE SYSTEM

To start the system, place diskette APPLE1: in the disk drive; then turn on the Apple's power. When the "WELCOME" message appears, Pascal is running. Using APPLE1: as the system diskette, you can file, edit, and execute previously compiled programs; but you cannot compile new programs. To change system diskettes, place APPLEØ: in the drive; then press the Apple's RESET key. Again, when the "WELCOME" message appears, Pascal is running. Using APPLEØ: as the system diskette, you can file, edit, compile, and execute programs; but you cannot start up the system from power-on.

FORMATTING NEW DISKETTES

To format a diskette, have Pascal's COMMAND prompt line showing. Place diskette APPLE3: in the disk drive, and type

X

In response to the query:

EXECUTE WHAT FILE?

type

APPLE3:FORMATTER

When the question:

FORMAT WHICH DISK ?

appears, place the new diskette in the disk drive, then type

4

and press the RETURN key. The diskette will be formatted. To leave the formatting program, press the RETURN key in response to the WHICH DISK question. A newly formatted diskette has the name BLANK:

COPYING DISKETTES

To copy a diskette, have the COMMAND prompt line showing, and put diskette APPLEØ: or APPLE1: in the disk drive. Get into the Filer by typing

F

When the FILER prompt line appears, put into the disk drive the source diskette to be copied. Then type

T

To the question:

TRANSFER ?

reply by typing the name of the source diskette to be copied, and then press the RETURN key. For example:

APPLE3:

To the next question:

TO WHERE ?

reply with the name of the destination diskette that is to become the backup copy. For example:

BLANK:

Then follow the instructions displayed on the screen, switching the diskettes back and forth until the copy is complete. Before you Quit the Filer, be sure to put your system diskette (usually APPLEØ:) back in the drive.

Note: you cannot make a copy onto a destination diskette that has the same name as the source diskette. Use the Filer to C(hange the name of either diskette, at least while making the copy.

EXECUTING A PROGRAM

To execute a previously compiled program, put your system diskette (APPLEØ: or APPLE1:) into the disk drive. With the COMMAND prompt line showing, enter the Filer by typing

F

When the FILER prompt line appears, put into the disk drive the diskette containing the program codefile that you wish to execute. Then type

T

for T(ransfer. To the question

TRANSFER ?

reply by typing the name of the program's diskette and codefile. For example,

APPLE3:GRAFDEMO.CODE

To the next question

TO WHERE ?

reply with the name of your system diskette, and the same filename (or another name, if you wish). For example,

APPLEØ:GRAFDEMO.CODE

When you are prompted

PUT IN APPLEØ:

follow the instructions, and press the spacebar. The program is then transferred onto your system diskette, which is where it must be in order to execute it. Now type Q to Q(uit the Filer, and when the COMMAND prompt appears, type X for X(ecute. When the Apple prompts

EXECUTE WHAT FILE?

answer by typing the name of your system diskette and the newly transferred codefile you wish to have executed. DO NOT type the .CODE suffix. In this example, you would type

APPLEØ:GRAFDEMO

The program should now run.

WRITING A PROGRAM

To start a new file in the Editor, put your system diskette (which must be APPLEØ: if you want to R(un your program) into the disk drive. With the COMMAND prompt line showing, type F to enter the Filer. Then type N for N(ew. If you are asked

THROW AWAY CURRENT WORKFILE ?

type Y for Y(es. When you see the message

WORKFILE CLEARED

type Q to Q(uit the Filer, and then type E to enter the Editor. This message appears:

>EDIT:
NO WORKFILE IS PRESENT. FILE? (<RET> FOR NO FILE <ESC-RET> TO EXIT)

Press the RETURN key, and the full EDIT: prompt line appears. You can now insert text at the cursor position by typing I for I(nsert and then typing your program. Conclude each insertion by pressing CTRL-C. Delete text at the cursor position by typing D for D(elete and then moving the cursor to erase text. Conclude each deletion by pressing CTRL-C . When you have written a version of your program, type Q to Q(uit the Editor, and then type U to U(pdate the workfile to contain your latest program version.

With the COMMAND prompt line showing, you can then type R to R(un your program. This automatically compiles the text workfile (using the Compiler program on APPLEØ:), stores the compiled code workfile, and executes it. To reenter the Editor, type E in response to the COMMAND prompt. The text workfile is automatically read back into the computer.

When a version of your program is complete, you can U(pdate the text workfile to contain that latest version and R(un the program to create a code workfile of that version. To save the workfile versions of your program on another diskette for later use, first save the workfile under another name on your system diskette (APPLEØ:). Type F in response to the COMMAND prompt to enter the Filer. Then type S for S(ave. When you see the prompt

SAVE AS ?

type the name of your system diskette and the filename under which you want your program saved. Do not type any .TEXT or .CODE suffix. For example, if you want your program saved under the filename DEMO , you might type

APPLEØ:DEMO

The text workfile SYSTEM.WRK.TEXT is saved as DEMO.TEXT on APPLEØ:,

and the code workfile SYSTEM.WRK.CODE is saved as DEMO.CODE .

Now you can T(ransfer those files to any other diskette, for safe keeping. Type

```
T
```

and when the Filer asks

```
TRANSFER ?
```

give the name of one of the S(aved files on your system diskette. In the previous example, you could type APPLEØ:DEMO.TEXT To the next question TO WHERE ? reply by typing the name of the diskette and file where you wish your program file to be stored. For example, you might type MYDISK:DEMO.TEXT The Apple will prompt you when it is time to put the destination diskette in the drive. When the text version of your program has been transferred onto the destination diskette, put your system diskette back in the drive. Now, type T for T(ransfer again, and transfer the code version of your program to the destination diskette in the same way you transferred the text version.

Remember to put APPLEØ: back in the disk drive before Q(uitting the Filer.

APPENDIX E

TWO-DRIVE STARTUP

This appendix is a tutorial session to get you started using the Language System with Pascal, on an Apple II with two or more diskette drives. If your system has only one diskette drive, please go back and read Appendix D instead.

EQUIPMENT YOU WILL NEED

You should have the following:

1. Your 48K Apple computer, with a Language Card installed, and at least two disk drives. The first two should be attached to a disk controller card in slot 6. All your disk controller cards should have the new PROMs, P5A and P6A, that came with the Language System.

2. A TV set or video monitor, connected to your Apple.

3. The following Language System diskettes:

    a. APPLE1:
    b. APPLE2:
    c. APPLE3:
    d. A blank diskette
    e. A second blank diskette

The diskette marked "APPLE1:" is needed to start the system. The diskette marked "APPLE2:" adds certain extra features to the system (the Compiler, the Assembler, and the Linker). You will not need the diskette marked "APPLE2:" until later. The diskette marked "APPLE3:" contains a number of useful utility programs. A diskette marked "APPLEØ:" is also included with the Language System. This diskette is normally used with single-drive systems.

The Apple and the TV or monitor should be plugged in. Turn on the TV now, so that it can warm up; but leave the Apple turned off.

MORE THAN TWO DISK DRIVES

If your system has more than two disk drives, the third drive gets connected to the "DRIVE 1" pins on the second controller, which goes in slot 5. A fourth drive is connected to the "DRIVE 2" pins on the second controller, in slot 5. A fifth and even a sixth drive can be connected to a controller in slot 4, using the "DRIVE 1" and "DRIVE 2" pins, respectively.

NUMBERING THE DISK DRIVES

Pascal assigns a "volume" number to each of the disk drives. It is not a bad idea to place tags with these numbers on your disk drives. Here's how the volume numbers are assigned to the various disk drives:

APPLE DISK DRIVE	PASCAL VOLUME
Slot 6, Drive 1	#4:
Slot 6, Drive 2	#5:
Slot 5, Drive 1	#11:
Slot 5, Drive 2	#12:
Slot 4, Drive 1	#9:
Slot 4, Drive 2	#1Ø:

You will find that you can refer to any diskette by either the name of the diskette (e.g., APPLE3:) or by the volume number of the drive in which it sits (e.g., #11:)

PASCAL IN SECONDS

Place the diskette marked "APPLE1:" in disk drive #4: (slot 6, drive 1). If you are not familiar with handling diskettes, see the manuals that came with your disk drives. Diskettes must be treated correctly if they are to last.

Close the door to disk drive #4: , and turn on the Apple. The rest is automatic. First, the message

APPLE II

appears at the top of your TV or monitor screen, and disk drive #4:'s "IN USE" light comes on. The disk drive emits a whirring, zickking sound that is as pleasant as a cat's purring, since it lets you know that everything is working. The screen lights up for an instant with a display of black at-signs (@) on a white background, then goes black again. Next the other disk drives are turned on, one at a time, as Apple Pascal finds out what is in each drive. A drive with no diskette in it may buzz and clatter a bit. When Apple Pascal cannot read anything from a disk drive, it recalibrates the drive's read-head

position (buzz, clatter) and then tries again. Now disk drive #4: stops entirely for a moment; then it whirrs some more. Finally, the message

```
WELCOME  APPLE1,  TO
U.C.S.D.  PASCAL SYSTEM II.1
CURRENT DATE IS 26-JUL-79
```

appears (the date will be different), followed in a second or so by a line at the top of the screen:

```
COMMAND: E(DIT, R(UN, F(ILE, C(OMP, L(IN
```

This line at the top of the screen is called a "prompt line". When you see this prompt line, you know that your Apple computer is running the Apple Pascal system.

Starting the system depends only on having APPLE1: in disk drive #4:. This time, you left the other drives empty; but you will soon discover that the system starts more quickly and quietly if the other drives have Pascal diskettes in them. For now, you could put diskettes APPLE2: and APPLE3: in any empty disk drives. Later, you will have other diskettes to put in them. In any case, make sure you never put two diskettes with the same name into the system at the same time. This may cause the directories of those diskettes to get scrambled.

CHANGING THE DATE

The date that comes on the diskette will not be correct. It is a good habit to reset the date the first time you use the Pascal System on any given day. It only takes a few seconds. Press F on the keyboard (without pressing the RETURN key or any other keys). The screen goes blank, and then this line appears at the top:

```
FILER: G, S, N, L, R, C, T, D, Q
```

This is a new prompt line. Prompt lines are named after their first word. The prompt line you first saw was the "COMMAND" prompt line. This one is the "FILER" prompt line. Sometimes we say that you are "in the Filer" when this line is at the top of the screen. Each of the letters on the prompt line represents a task that you can ask the Apple to do. For example, to change the date, press D (again, just type the single key, without pressing RETURN or any other keys).

When you do, another message is put on the screen. It says:

```
DATE SET: <1..31>-<JAN..DEC>-<ØØ..99>
TODAY IS 26-JUL-79
NEW DATE ?
```

It doesn't really mean that today is 26-JUL-79 (or whatever date your screen shows), but that the Apple THINKS that is today's date. Since it isn't, you can change the date to be correct. The correct form for typing the date is shown on the second line of the message: one or two digits giving the day of the month, followed by a minus sign, followed by the first three letters of the name of the month, followed by another minus sign, followed by the last two digits of the current year. Then press the key marked RETURN .

If the month and year are correct (as they will often be, when you change the date) all you have to do is type the correct day of the month, and press the RETURN key. The system will assume that you mean to keep the same month and year displayed by the message. If you type a day and a month, the system will assume you mean to keep only the year the same.

Go ahead and make the date correct. This is your first interaction with the system, and is typical of how the system is used. In general, at the top of the screen there will usually be a prompt line which represents several choices of action. When you type the first letter of one of the choices, either you will be shown a new prompt line giving a further list of choices, or else the system will carry out the desired action directly. If you type a letter that does not correspond to one of the choices, the prompt line blinks but otherwise nothing happens. Remember to type only a single letter to indicate your choice; it is not necessary to press the RETURN key afterward.

Sometimes, as when setting the date, you are asked to type a response of several characters. You tell the system that your response is complete by pressing the RETURN key. If you make a typing error before pressing the RETURN key, you can back up and correct the error by pressing the left-arrow key. You should experiment by making deliberate errors in entering a date, and then erasing the errors with the left-arrow key.

One further note. Normally, your new date is saved on diskette APPLE1:, so the system "remembers" this date the next time you turn the Apple on. However, since you are using the write-protected diskettes that came with your Language System, your new date was not permanently saved. The next time you turn the Apple off, the new date will be "forgotten". By the end of this session, you will have made backup copies of the Language System diskettes. From then on, you will use these copies, which are not write-protected, and your date changes will be saved.

MAKING BACKUP DISKETTE COPIES

WHY WE MAKE BACKUPS

Ask yourself this question: What would happen to your system if you were to lose or damage one of the system diskettes (APPLE1:, APPLE2:, or APPLE3:)? It would be as bad as losing your Apple itself, as far as your being able to use Apple Pascal.

These diskettes are quite precious. The first thing you should do, therefore, is to make backup copies of them. Afterward, you should never use the originals, but put them someplace where the temperature is moderate, where there is no danger of them getting wet, and where such diskette destroyers as dogs, dirt, children, and magnetic fields cannot get at them.

A truly cautious person will keep on hand two backup copies of each original. That way, you will need to use an original only in the very rare case when both of its backup copies are lost (when one copy is lost or damaged, another backup copy is made from the surviving backup copy). If your backups were damaged or erased while in use, find out why they were destroyed before inserting your only surviving copy. Using diskettes for which you have backups, repeat the procedure that destroyed the first diskettes, and if you can't figure out what the problem is, bring your system to the dealer to make sure it is working correctly.

HOW WE MAKE BACKUPS

The Pascal system can copy all the information from one diskette (or any portion of the information) onto another diskette. But the system cannot store information on a new diskette, just as that diskette comes from the computer store. Therefore, the system is supplied with a program that allows you to take any 5-inch floppy diskette and "format" it so that it will work with the Apple Pascal system.

Incidentally, this is one of the nice little things about the Apple system: ANY high-quality 5-inch floppy diskette (Apple recommends diskettes made by Dysan Corporation) will work on it. Some systems require you to have "1Ø sector" or "15 sector" or "soft sectored" diskettes. The Apple doesn't care, it takes any of these kinds of diskettes, and (through the FORMATTER program) makes them into the kind of diskette it needs.

If you have been following this session by carrying out the instructions on your Apple, the FILER prompt line should be showing at the top of the screen:

```
FILER: G, S, N, L, R, C, T, D, Q
```

Type Q on the keyboard to Quit the Filer.

GETTING THE BIG PICTURE

When you Quit the Filer, disk drive #4: whirrs, and you see the COMMAND prompt line again:

```
COMMAND: E(DIT, R(UN, F(ILE, C(OMP, L(IN
```


There is actually more of this prompt line, off to the right of your TV or monitor. To see the rest of the screen, hold down the key marked CTRL and, while holding it, press the "A" right alongside it. (Or, to be brief, we say: "press CTRL-A".)

You now see

```
K, X(ECUTE, A(SSEM, D(EBUG,?
```

This is simply the rest of the line that began "COMMAND:". All together, the full prompt line would look like this:

```
COMMAND: E(DIT, R(UN, F(ILE, C(OMP, L(INK, X(ECUTE, A(SSEM, D(EBUG,?
```

The Apple Pascal system displays information on a "screen" that is 8Ø characters wide, but your TV or monitor shows only the leftmost 4Ø characters or the rightmost 4Ø characters at any one time. You use the CTRL-A trick whenever you wish to see if there is more stuff on the other "half" of the screen. Repeated pressing of CTRL-A flips back and forth between the left half of the screen and the right half.

Also, sometimes the TV display will seem to be blank. This might mean that you are just staring at the empty right half of the screen. Before you come to the conclusion that something is wrong, always try CTRL-A. You get back to the left side of the screen by typing CTRL-A again, and you might find that everything is OK after all.

Summary of this diversion: The screen is really twice as wide as it looks. To flip from the left side to the right side or back again, you type CTRL-A.

FORMATTING NEW DISKETTES

Place diskette APPLE3: in any available disk drive except drive #4: . This has to be done because the FORMATTER program is on APPLE3:. Now, with the COMMAND prompt line at the top of the screen, type

```
X
```

and the screen responds:

```
EXECUTE WHAT FILE?
```

You type

```
APPLE3:FORMATTER
```

and press the key marked RETURN .

The disk drive containing APPLE3: whirrs a bit and the screen says:

```
APPLE DISK FORMATTER PROGRAM
FORMAT WHICH DISK (4, 5, 9..12) ?
```

Take all the new, blank diskettes that you are going to use with the Pascal System (but not, of course, any diskettes that have precious information on them, such as the diskettes that came with the Pascal System) and place them in a pile. Their labels should be blank. Make sure that you don't have any diskettes with data in a non-Pascal format, such as BASIC diskettes: the Pascal system will be unable to read them, and will regard them as blank, erasing any old information in the formatting process.

Remove the diskette in disk drive #5: (if yours is a two-drive system, you will be removing diskette APPLE3:) and put one of the new, blank diskettes into that drive. Then type

```
5
```

and press the key marked RETURN .

If the diskette in drive #5: has already been formatted, you will receive a warning. For example, if you have left APPLE3: in that drive you will be warned with the message

```
DESTROY DIRECTORY OF APPLE3 ?
```

At this point you can type

```
N
```

(which stands for "No") without pressing the RETURN key, and your diskette will not be destroyed. Let's assume that you have a new, unformatted diskette. Then you will not get any warning, but the Apple will place this message on the screen:

```
NOW FORMATTING DISKETTE IN DRIVE 5
```

Disk drive #5: will make some clickings and buzzings and begin to whirr and zick. The process takes about 32 seconds. When formatting is complete, the screen again shows the message

```
FORMAT WHICH DISK (4, 5, 9..12) ?
```

Now you have a formatted diskette. We suggest that you write "Pascal" in small letters at the top of the diskette's label, using a marking pen. Do not use a pencil or ballpoint pen, as the pressure of writing may damage the diskette. The label will let you know that the diskette is formatted for use with the Apple Pascal system, and you can distinguish it from unformatted diskettes, BASIC diskettes, or diskettes for use with other systems.

While you are at it, repeat this formatting process on all the new diskettes that you want to use with the Apple Pascal System. With each new diskette, place it in drive #5: , type 5 and press the RETURN key.

Note: If you have more than two drives, you can simplify the procedure by putting the next diskette to be formatted into any unoccupied drive. Then, when the system asks

```
FORMAT WHICH DISK (4, 5, 9..12) ?
```

just type the correct volume number of the drive containing your new, blank diskette, and then press the RETURN key. This will save you some diskette-swapping.

When you have finished formatting all your new diskettes, and have written the word "Pascal" on each of them, answer the question

```
FORMAT WHICH DISK (4, 5, 9..12) ?
```

with a simple press of the key marked RETURN . You get the message

```
THAT'S ALL FOLKS...
```

And if you watch the top of the screen, the line

```
COMMAND: E(DIT, R(UN, F(ILE, C(OMP, L(INK, X(ECUTE, A(SSEM, D(EBUG,?
```

appears (of course, it doesn't all appear; but you know it's there, and can check with CTRL-A).

MAKING THE ACTUAL COPIES

As you have seen, you can get into the Filer by typing F when you have the COMMAND prompt line on the screen. You must have diskette APPLE1: or diskette APPLEØ: in one of the disk drives when you type F to enter the Filer. If you forget (and APPLE1: is your system diskette), you will get the message

```
NO FILE APPLE1:SYSTEM.FILER
```

If this happens, just put APPLE1: in any drive and type F again.

The Filer is that portion of the system which allows you to manipulate information on diskettes. One of the Filer's abilities is to transfer information from one diskette to another. To invoke this facility, once you have the FILER prompt line on the screen, type T for T(ransfer.

This is what you see:

```
TRANSFER ?
```

Let's say that you want to make a backup copy of diskette APPLE3: , by copying APPLE3: onto one of your newly formatted diskettes. Put APPLE3: into any available disk drive, and put a newly formatted diskette into any other drive. If your system has only two drives, you will have to remove diskette APPLE1: from drive #4: . Once the FILER prompt line is showing, APPLE1: is no longer needed until you wish to Quit the Filer and return to the COMMAND prompt line. Now, answer the question by typing the name of the source diskette to be copied:

```
APPLE3:
```

When you press the RETURN key, the computer checks to see that diskette APPLE3: is in one of the disk drives. If it is not, you will see the message

```
APPLE3:
NO SUCH VOL ON-LINE <SOURCE>
```

In that case, just put APPLE3: in a disk drive and type T for Transfer again. If the computer succeeds in finding APPLE3:, it asks you the next obvious question: If you are going to transfer something, then

```
TO WHERE ?
```

Answer this question by typing the name of the diskette that is to become an exact backup copy of APPLE3:

```
BLANK:
```

Remember that BLANK: is the name given to all newly formatted diskettes by the FORMATTER program. The colons (:) that appear after the diskette names are quite significant: they indicate that the entire diskette is being referred to.

After you have told the computer where you want APPLE3:'s information transferred (and pressed the key marked "RETURN"), it checks to see that BLANK: is also in one of the disk drives. If it is not, you will see the message

```
PUT IN BLANK:
TYPE <SPACE> TO CONTINUE
```

In that case, put BLANK: into any disk drive except the one containing APPLE3:, and press the Apple's spacebar. When the computer succeeds in finding both the source and the destination diskettes, it says

```
TRANSFER 280 BLOCKS ? (Y/N)
```

This message is mainly there to give you a chance to abandon the transfer if you made a typing error in the names of the source or the destination diskettes. The phrase "280 BLOCKS" means merely "THE WHOLE DISKETTE". In any case, you type

```
Y
```


All the information on diskette APPLE3:, including the diskette's name, will be copied onto diskette BLANK:, completely overwriting BLANK:. Therefore, the computer warns you that you are about to lose any information that might be stored on BLANK:. It says

```
DESTROY BLANK: ?
```

Since you want to turn BLANK: into a perfect copy of APPLE3:, the answer is

```
Y
```

The process is under way. It takes about two minutes to copy and check the entire diskette. When copying is done the screen celebrates by saying:

```
APPLE3:                    --> BLANK:
```

by which cryptic remark the computer is telling you that the contents of APPLE3:, including the diskette's name, have been copied onto the diskette that used to be called BLANK:. This is just what you wanted.

There are now two diskettes with the same name, both in the system at once. This is a risky situation, confusing both to you and to the computer, so be sure to remove the new copy right away. Now, using a marking pen, write "APPLE3:" on the new diskette's label. Do not use a pencil or a ballpoint pen, as the pressure of writing may damage the diskette. It is very important to label diskettes immediately, so you know what information is stored on them.

DO IT AGAIN, SAM

You should, at this time, make sure that you have at least one backup copy of each of your system diskettes: APPLE1:, APPLE2:, and APPLE3:. In each case, just place the source diskette to be copied from in one drive, the blank destination diskette to be copied onto in another drive, and then type T to begin the Transfer. While you are at it, make a backup copy of APPLE0: , too. It may come in handy, later on.

BEFORE you type Q to Quit the Filer and return to the COMMAND prompt line, be sure to put diskette APPLE1: back into drive #4: If you forget to do this, the computer will stop responding to its keyboard after you type Q ; even the RESET key will have no effect. You will have to turn the computer off, put APPLE1: in drive #4:, and turn the computer on again.

Finally, you should store the original diskettes (and one extra copy, if you like to be really safe) away, in a safe place.

USING THE SYSTEM

A DEMONSTRATION

At last, a reward for all your work to this point: you are finally ready to use the Apple Pascal system to run a program. Diskette APPLE3: contains several "demonstration" programs. To see a list of those programs, put APPLE3: in any disk drive except #4: (APPLE1: must be in drive #4:). Now, enter the Filer by typing F in response to the COMMAND prompt line. When the FILER prompt line appears on the screen, type L to List a diskette's directory. The Filer says:

```
DIR LISTING OF ?
```

In response, type the name of the diskette whose directory you wish to see:

```
APPLE3:
```

A long list of program files now appears on the screen, many of them both in their .TEXT versions (the form in which they are written and edited) and also in their compiled .CODE versions (the form in which they can be executed). When the screen is full, the display stops and the message

```
TYPE <SPACE> TO CONTINUE
```

appears at the top of the screen. Press the Apple's spacebar to see the remaining files. For now, we are interested in the file named GRAFDEMO.CODE .

Since the system diskette APPLE1: is already in disk drive #4: , you may now type Q to Quit the Filer. When the COMMAND prompt line appears, type X for X(ecute. The computer says

```
EXECUTE WHAT FILE?
```

Answer by typing the name of the diskette and file you wish to have executed:

```
APPLE3:GRAFDEMO
```

Note: DO NOT type the suffix .CODE ; the system knows you can execute only a code file, so it automatically supplies the suffix .CODE for you, in addition to any name that you type.

When this message appears

```
PRESS ANY KEY TO QUIT.
PLEASE WAIT WHILE CREATING BUTTERFLY
```

the program is running. After a short pause, the display begins. Just sit back and enjoy it: soon you'll be writing your own programs using these and other features of Apple Pascal. When you are tired of watching, press the spacebar on the Apple's keyboard to return to the COMMAND prompt line. You can use this same procedure to run any of the programs on APPLE3: . These programs are discussed in Appendix A.

DO IT YOURSELF

Now, for some more experience at using the Apple Pascal system, let's try writing a short program. This discussion will assume that you are using your new copies of the Pascal diskettes. You should be using a new copy of APPLE1: as your system diskette (or "boot diskette" as it is often called). This copy is not write-protected, and you have never used the Editor to create any new files on it before. Put the new copy of APPLE1: in the boot drive, volume #4: . You should also put a copy of APPLE2: in any other drive (APPLE2: contains the Compiler program).

With the COMMAND prompt line showing, type E to select the E(dit option. Soon, this message appears:

```
>EDIT:
NO WORKFILE IS PRESENT. FILE?( <RET> FOR NO FILE <ESC-RET> TO EXIT )
:
```

As usual, you must use CTRL-A to see the right half of the message. This message gives you some information and some choices. The first word, >EDIT: , tells you that you are now in the Editor. The next sentence, NO WORKFILE IS PRESENT , tells you that you have not yet used the Editor to create a "workfile", which is a "scratchpad" diskette copy of a program you are working on. If there had been a workfile on APPLE1: , that file would have been read into the Editor automatically.

Since there was no workfile to read in, the Editor asks you, FILE? If you now typed the name (including the drive's volume number or the diskette's name) of a .TEXT file stored on APPLE1: or on APPLE2:, that textfile would be read into the Editor. However, there are no .TEXT files on APPLE1: or APPLE2: yet, and besides, you want to write a new program. In parentheses, you are shown how to say that you don't want to read in an old file: <RET> FOR NO FILE . This means that, if you press the Apple's RETURN key, no file will be read in and you can start a new file of your own. That's just what you want to do, so press the Apple's RETURN key (the rest of the message says if you first press the ESC key and THEN press the RETURN key, you'll be sent back to the

COMMAND prompt line). When you have pressed the RETURN key, the full EDIT prompt line appears:

```
>EDIT: A(DJST C(PY D(LETE F(IND I(NSRT ...
```

The chapter called THE EDITOR in the Apple Pascal Operating System Reference Manual explains all of these command options in detail; for now you will only need a few of them. The first one you will use is I(NSRT , which selects the Editor's mode for inserting new text. Type I to select Insert mode, and yet another prompt line appears:

```
>INSERT: TEXT [<BS> A CHAR,<DEL> A LINE] [<ETX>ACCEPTS, <ESC>ESCAPES]
```

As long as this line is showing at the top of the screen anything you type will be placed on the screen, just to the left of the white square "cursor". If the cursor is in the middle of a line, the rest of the line is pushed over to make room for the new text. If you make a mistake, just use the left-arrow key to backspace over the error, and then retype. At any time during an insertion, if you press the Apple's ESC key your insertion will be erased. At any time during an insertion, if you press CTRL-C the insertion will be made a permanent part of your file, safe from being erased by ESC or by the left-arrow key. You can then type I to reenter Insert mode and type more text.

Now for our program. With the INSERT prompt line showing, press the RETURN key a couple of times, to move the cursor down, and then type

```
PRORAFM DEMO;
```

You can use any name for your program, but in this discussion it will be called DEMO . Now press CTRL-C (type C while holding down the CTRL key). Your insertion so far is made "permanent", and the EDIT prompt line reappears. But, horrors! You made several typing errors when typing the word PROGRAM . Since you have already pressed CTRL-C , it is too late to backspace over your errors and retype them.

Fortunately, there are other ways. First, let's correct the missing G in PROGRAM . Using the left-arrow key, move the cursor left until it is sitting directly on the R . Then type I to reenter Insert mode. Ignore the fact that the remainder of the line seems to have suddenly disappeared, and type the missing letter G . When you press CTRL-C to make this insertion permanent, the rest of the line returns:

```
PROGRAFM DEMO;
```

The letter F is certainly not needed, so move the cursor right (using the right-arrow key) until it is sitting directly on the F . Now type D to select the Editor's D(LETE option. When the DELETE prompt line appears,

```
>DELETE: < > <MOVING COMMANDS> [<ETX> TO DELETE, <ESC> TO ABORT]
```

press the right-arrow key once. The offending F instantly disappears. In Delete mode, moving the cursor in any direction deletes text. If you move the cursor back again, the deleted text reappears. What happens next is similar to Insert mode: if you press the ESC key, the deletion is forgotten, as if it had never happened. If you press CTRL-C, the deletion is made a permanent part of your file. To remove that F permanently, press CTRL-C. The line closes in to fill the deleted letter's place:

```
PROGRAM DEMO;
```

Now you know how to use the Editor's Insert and Delete modes to write text and to correct your errors. Try typing the rest of program DEMO into your file. Be sure to "accept" your insertions, from time to time, by pressing CTRL-C . That way, you minimize your loss if you accidentally press the ESC key. Here is the complete program:

```
PROGRAM DEMO;

  USES TURTLEGRAPHICS, APPLESTUFF;
  VAR ANGLE, DISTANCE : INTEGER;

  PROCEDURE CRAWL;
  BEGIN
    MOVE (2 * DISTANCE);
    TURN (ANGLE)
  END;

  BEGIN
    ANGLE := Ø;
    REPEAT
      INITTURTLE;
      PENCOLOR (WHITE);
      FOR DISTANCE := 1 TO 99 DO CRAWL;
      ANGLE := ANGLE + 5
    UNTIL KEYPRESS;
    TEXTMODE
  END.
```

When you are typing this program, the punctuation and spelling must be exactly as shown. The indentation of the lines is not important, but it easier to read as shown. You will notice that, once you have started a new indentation, the Editor maintains that indentation for you. To move back to the left, just press the left-arrow key before you type anything on the new line.

Program DEMO makes use of graphics routines in the Unit TURTLEGRAPHICS, and uses the keypress function from the Unit APPLESTUFF (see Chapter 7 for details). The third line of the program declares two integer variables, DISTANCE and ANGLE. Next, a Pascal procedure named CRAWL is defined, between the first BEGIN and END; . From here on, each time this new Pascal statement CRAWL is used, a graphics "turtle" will trace a line on the screen, of length 2*DISTANCE moving in the current direction, and will then change the direction by an amount ANGLE.

The next BEGIN and the last END. outline the main program. The portion of the program from REPEAT to UNTIL KEYPRESS is repeated over and over again, until any key on the Apple's keyboard is pressed.

In each repetition, the screen is cleared and the tracing color is set to WHITE. Then the procedure CRAWL is performed, first with the value of DISTANCE set to one, then with DISTANCE set to the value two, and so on, until DISTANCE is set to 99 . The "turtle" moves, then turns, then moves some more, then turns again, and so on, for 99 steps. That completes one design on the screen. In the next repetition, if no key has been pressed, the ANGLE has increased by 5 degrees, the screen is cleared by INITTURTLE, and the whole process starts again.

Now you should save this program. With the EDIT prompt line showing, type Q to select the Q(UIT option. The following message appears:

```
>QUIT:
     U(PDATE THE WORKFILE AND LEAVE
     E(XIT WITHOUT UPDATING
     R(ETURN TO THE EDITOR WITHOUT UPDATING
     W(RITE TO A FILE NAME AND RETURN
```

Type U to create a "workfile" diskette copy of your program (future versions of this file will be "Updates)". This workfile is a file on your boot diskette (APPLE1:) called SYSTEM.WRK.TEXT . The computer says

```
WRITING..
YOUR FILE IS 33Ø BYTES LONG.
```

(the number of bytes may be a little different) and then the COMMAND prompt line reappears. Now type R to select the R(UN option. This automatically calls the Compiler for you, since the workfile contains text. The disk drive containing APPLE2: whirrs and, if you have typed the program perfectly, the following messages (again, perhaps with slightly different numbers) appear, one by one:

```
COMPILING...


PASCAL COMPILER II.1 [B2B]
<   Ø>....
TURTLEGR [ 2483 WORDS]
<   5>.........................
APPLESTU [ 1Ø78 WORDS]
<  3Ø>...................
CRAWL    [ 1Ø98 WORDS]
<  46>.....
DEMO     [ 11Ø9 WORDS]
<  51>.........
59 LINES
SMALLEST AVAILABLE SPACE = 1Ø98 WORDS
```

If the Compiler discovers mistakes, it will give you a message such as

```
PROFRAM <<<<
LINE 2, ERROR 18: <SP>(CONTINUE), <ESC>(TERMINATE), E(DIT
```

Don't despair; just type E for E(DIT . Your workfile will be automatically read back into the Editor for repairs. Read the error message at the top of the screen, press the spacebar, and make any necessary changes using I(nsert and D(elete. Then Q(uit, U(pdate the workfile, and R(un your program again, by typing Q U R (the Apple will store up several commands in advance).

When your program has been successfully Compiled, it is automatically executed. You will see the message

```
RUNNING...
```

and then a horizontal line appears on the screen. That is the first design your program draws: the white "turtle" moves out a distance 2*1 , turns an angle Ø ; moves 2*2 , turns Ø ; moves 2*3 , turns Ø ; etc. Keep watching as successive designs turn through larger and larger angles between moves. When you want to interrupt the program, press any key on the keyboard. You can R(un the program again at any time, by typing R . Since the latest version of your program has already been compiled, it will be executed immediately, this time.

Try making changes to the program, by setting a different starting ANGLE, or a different increment to the ANGLE, or a different distance to MOVE. To do this, type E for E(DIT, use I(nsert and D(elete to make changes, and then Q(uit, U(pdate the workfile, and R(un again by typing Q U R . This cycle of Edit-Run-Edit-Run is the basis of all program development in the Apple Pascal system.

The workfile on APPLE1: now contains the text version of your program in a file named SYSTEM.WRK.TEXT , and the compiled P-code version of your program in another file named SYSTEM.WRK.CODE . When your program is running as you want it to, you should save the text and code workfile under other filenames. With the COMMAND prompt line showing, type F to enter the Filer. When the FILER prompt line appears, place in any available drive the diskette on which you want your program stored. Then type S for S(ave. You will be asked

```
SAVE AS ?
```

and you should respond by typing the name of the destination diskette, followed by a colon, followed by any filename with ten or fewer characters. For example, you might type

```
MYDISK:DEMO
```

When you press the RETURN key, the boot diskette's workfile, SYSTEM.WRK.TEXT and SYSTEM.WRK.CODE, is saved on MYDISK: under the

filenames DEMO.TEXT and DEMO.CODE . These messages will tell you what has happened:

```
APPLE1:SYSTEM.WRK.TEXT
 --> MYDISK:DEMO.TEXT
APPLE1:SYSTEM.WRK.CODE
 --> MYDISK:DEMO.CODE
```

WHAT TO LEAVE IN THE DRIVES

When you turn the Apple off, it is a good idea to leave the diskette called APPLE1: in disk drive #4: . If there is no diskette or some other diskette in #4: when the Apple is accidentally turned on, the drive will spin the disk indefinitely. If this continues for hours and hours, some wear will take place on the diskette and the drive. So, it is a good idea to make a habit of leaving a copy of APPLE1: (now that you have copies) in #4: when you turn the system off.

Of course, if you turn on the system and APPLE1: is not in #4:, just press the key marked RESET . Place APPLE1: in #4: and turn the system off and then on again. No damage results from turning on the computer with the wrong diskette (or no diskette) in the drive. Gradual, unnecessary wear results from leaving the disk drive running for a long period of time with the incorrect diskette (or no diskette) in the drive.

USING MORE THAN TWO DRIVES

The primary difference between using a two-drive system and using larger systems is that you rarely need to remove APPLE1: from its usual location in drive #4: , and can do all copying and transfering between files in the other drives.

For example, with four drives, you can have APPLE1: in #4:, APPLE2: in #5:, and APPLE3: in #11:; then you can format diskettes by placing them in #12:, without having to remove any of the system diskettes.

A one-drive system is a useful tool for learning Pascal and running programs written on other systems. A one-drive system can, in fact, do anything that the larger systems can do, up to the limits of the actual storage space available. For software development of any magnitude, however, two drives are recommended. Again, more drives make life easier. Word processing, using the text editor, is most pleasant with a three-drive system. Some business applications, which can benefit from having over half a megabyte on line, might use six drives.

No specific instructions will be given here on using multiple-drive systems. Acquaintance with a two-drive system should be sufficient introduction.

MULTIPLE-DRIVE SUMMARY

STARTING UP THE SYSTEM

To start the system, place diskette APPLE1: in disk drive #4: (slot 6, drive 1); then turn on the Apple's power. When the "WELCOME" message appears, Pascal is running.

FORMATTING NEW DISKETTES

To format a new diskette, have Pascal's COMMAND prompt line showing. Place diskette APPLE3: in any drive except #4: , and type

```
X
```

Now, in response to the query

```
EXECUTE WHAT FILE?
```

type

```
APPLE3:FORMATTER
```

When the question:

```
FORMAT WHICH DISK ?
```

appears, place the new diskette in any drive except #4: , and then type the number of that drive. For example, if you put the new diskette in drive #5: , type

```
5
```

When you press the RETURN key, the diskette will be formatted. To leave the formatting program, press the RETURN key in response to the question WHICH DISK ? A newly formatted diskette has the name BLANK:

COPYING DISKETTES

To copy a diskette, have the COMMAND prompt line showing, and put APPLE1: in drive #4: . Get into the Filer by typing

```
F
```

Once the FILER prompt line is showing, you may remove APPLE1: from its drive if you need to. Put the source diskette you wish to copy into one drive, and the destination diskette you want to copy onto into another drive, then type

```
T
```

Now, when this question appears:

```
TRANSFER ?
```

reply by typing the name of the source diskette to be copied, and then press the RETURN key. For example, you might type

```
APPLE3:
```

Now, when the next question appears:

```
TO WHERE ?
```

reply with the name of the destination diskette that is to become the backup copy. For example, you might type

```
BLANK:
```

Lastly, you will be asked

```
TRANSFER 280 BLOCKS ?
```

and

```
DESTROY BLANK: ?
```

Reply

```
Y
```

to both, and BLANK: will be turned into a perfect copy of APPLE3: . Be sure to put diskette APPLE1: back into drive #4: before Q(uitting the Filer.

EXECUTING A PROGRAM

To execute a previously compiled program, put APPLE1: in drive #4: and put the diskette containing the program file into any other drive. With the COMMAND prompt line showing, type X for X(ecute. When the computer prompts

```
EXECUTE WHAT FILE?
```

answer by typing the name of the diskette and codefile you wish to have executed. DO NOT type the .CODE suffix. For example, to execute the program GRAFDEMO.CODE on diskette APPLE3: , you would type

```
APPLE3:GRAFDEMO
```

The program should now run.

WRITING A PROGRAM

To start a new file in the Editor, put APPLE1: in drive #4: and put APPLE2: in drive #5: . With the COMMAND prompt line showing, type F to enter the Filer. Then type N for N(ew. If you are asked

```
THROW AWAY CURRENT WORKFILE ?
```

type Y for Y(es. When you see the message

```
WORKFILE CLEARED
```

type Q to Q(uit the Filer, and then type E to enter the Editor. This message appears:

```
>EDIT:
NO WORKFILE IS PRESENT. FILE? ( <RET> FOR NO FILE <ESC-RET> TO EXIT )
```

Press the RETURN key, and the full EDIT: prompt line appears. You can now insert text at the cursor position by typing I for I(nsert and then typing your program. Conclude each insertion by pressing CTRL-C. Delete text at the cursor position by typing D for D(elete and then moving the cursor to erase text. Conclude each deletion by pressing CTRL-C . When you have written a version of your program, type Q to Q(uit the Editor, and then type U to U(pdate the workfile to contain your latest program version.

With the COMMAND prompt line showing, you can then type R to R(un your program. This automatically compiles the text workfile (using the Compiler program on APPLE2:), stores the compiled code workfile, and executes it. To reenter the Editor, type E in response to the COMMAND prompt. The text workfile is automatically read back into the computer.

When a version of your program is complete, you can U(pdate the text workfile to contain that latest version and R(un the program to create a code workfile of that version. To save the workfile versions of your program on another diskette for later use, place that diskette in drive #5: and type F in response to the COMMAND prompt to enter the Filer. Then type S for S(ave. When you see the prompt

```
SAVE AS ?
```

type the name of the diskette and file where you want your program saved. Do not type any .TEXT or .CODE suffix. For example, if you want your program saved under the filename DEMO on the diskette MYDISK: , you might type

```
MYDISK:DEMO
```

The text workfile SYSTEM.WRK.TEXT on APPLE1: is saved as DEMO.TEXT on MYDISK:, and the code workfile SYSTEM.WRK.CODE is saved as DEMO.CODE on MYDISK: .

APPENDIX F
APPLE PASCAL SYNTAX

These diagrams represent all of the syntax of Apple Pascal. However, they do not show the semantic rules. To understand the distinction between syntax and semantics, consider the sentence "John Smith is a citizen of the three of clubs." This sentence is correct syntactically (i.e., grammatically) but wrong semantically -- the three of clubs is not something one can be a citizen of.

Similarly, the diagram for a statement shows that one kind of statement is an identifier optionally followed by one or more expressions in parentheses. The diagram does not show the semantic restriction, which is that the identifier must be the identifier of a procedure. Some of the important semantic restrictions are given in the notes accompanying the diagrams.

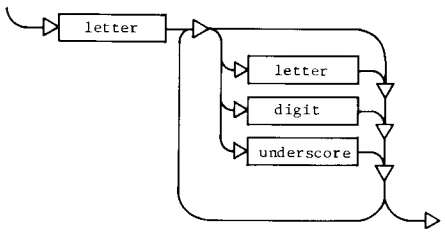
With this limitation in mind, you will find that the diagrams are useful as reference material. To read one of these diagrams, start at the left and follow arrows until you come out at the right. Whenever the arrows branch, you can go either way. Any path that goes through from the left to the right defines a syntactically correct Apple Pascal construction.

Circles and ovals are used to enclose characters and words that are to be typed exactly as shown; for example, the word NIL in the diagram for an unsigned constant. Boxes with square corners enclose words and phrases that stand for something else; for example, the word "letter" in the diagram for an identifier stands for any letter.

The vertical arrow symbol used in these diagrams corresponds to the "^" character in the text of this document and on the Apple keyboard.

A word or phrase that you find in a square-cornered box is the title of another diagram; the diagram shows what the word or phrase can stand for when it appears in other diagrams. (Exceptions: there are no diagrams for "letter," "digit," and "underscore.")

identifier

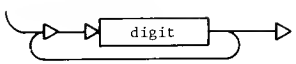


1. The letters are a..z and A..Z .

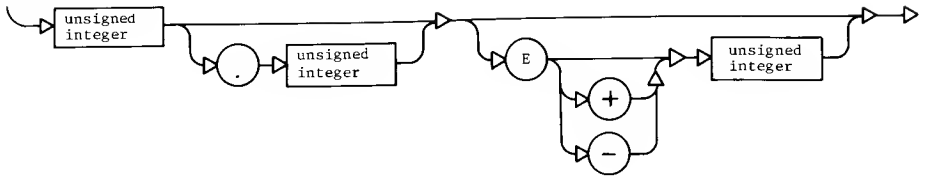
2. The digits are Ø..9 .

3. The underscore character, _ , is not available on the Apple keyboard. However some external terminals provide it.

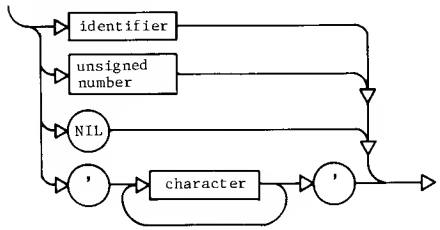
unsigned integer



unsigned number



unsigned constant



1. The identifier in this diagram must be the identifier of a constant.

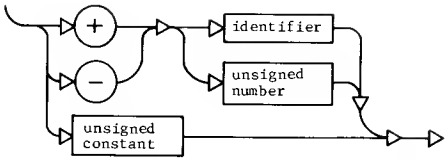
2. The bottom line of the diagram represents a string constant. A single apostrophe cannot appear as a character in the string constant, since this would end the constant. However, you can place two consecutive apostrophes in the string constant, and the result will be a single apostrophe in the value of the string. For example:

```
WRITELN('DON''T FORGET TO BOOGIE!')
```

will cause the following output:

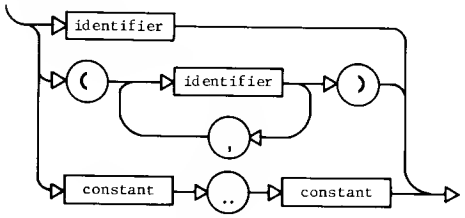
```
DON'T FORGET TO BOOGIE!
```

constant



The identifier in this diagram must be the identifier of a constant.

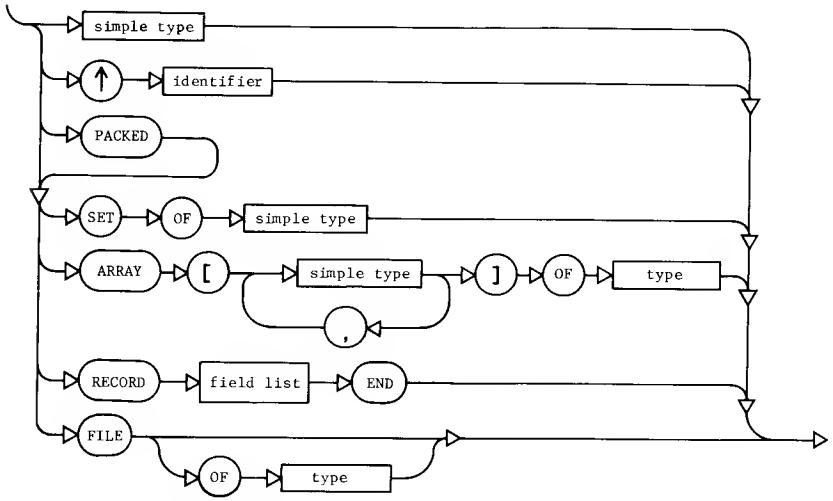
simple type



1. The identifier in the top line of this diagram must be the identifier of a type.

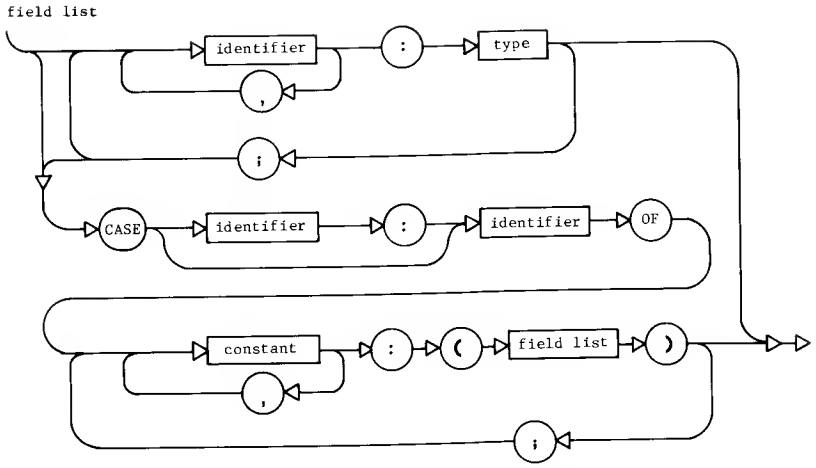
2. The identifier(s) in the second line define a scalar type. They are being declared, so they must be identifiers that are not yet declared or predefined.

type



The identifier in this diagram must be the identifier of a type.

field list

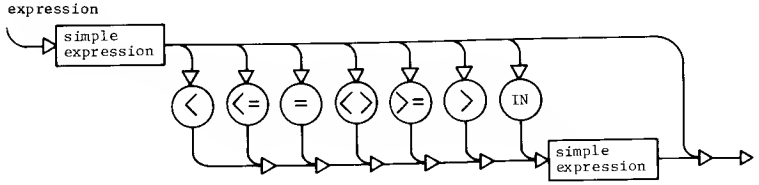


1. The identifier(s) in the top line are being declared, so they must be identifiers that are not yet declared or predefined.

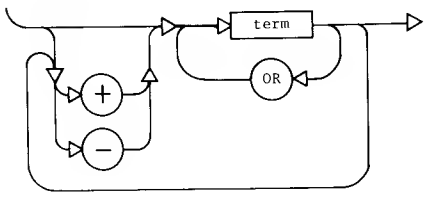
2. The identifier between the word CASE and the colon is the tag field. It is being declared, so it must be an identifier that is not yet declared or predefined.

3. The identifier between the colon and the word OF must be the identifier of a type.

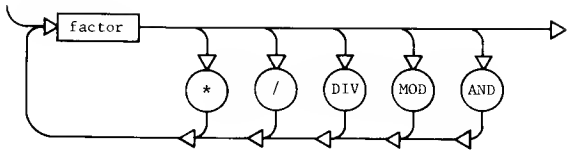
expression



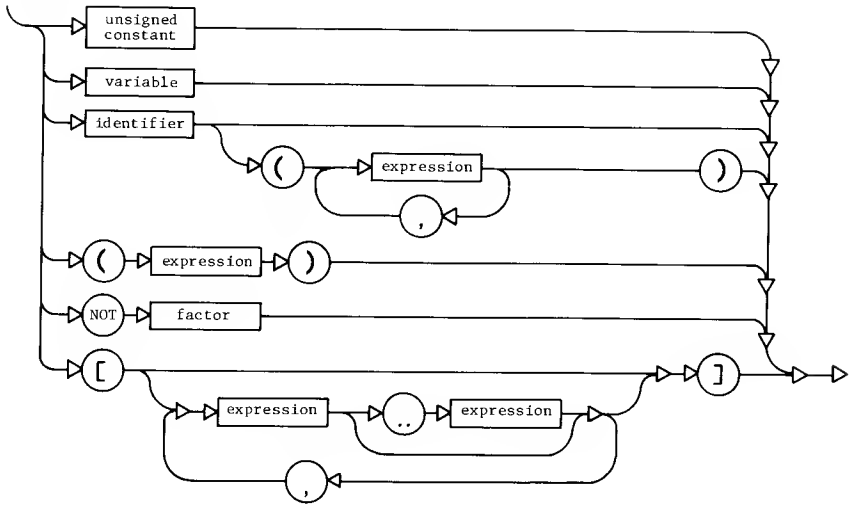
simple expression



term



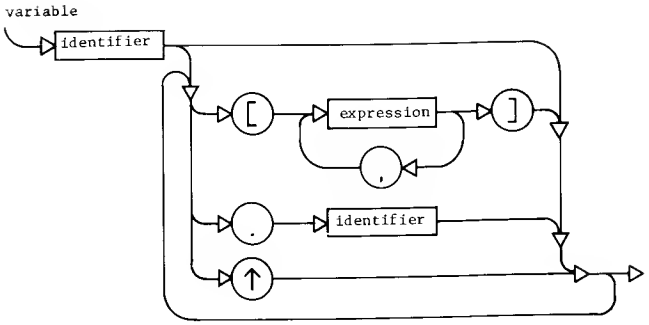
factor



1. The identifier in this diagram must be the identifier of a function.

2. The bottom portion of the diagram (square brackets and expressions) indicates the formation of a set. The values of the expressions must be of the same underlying type.

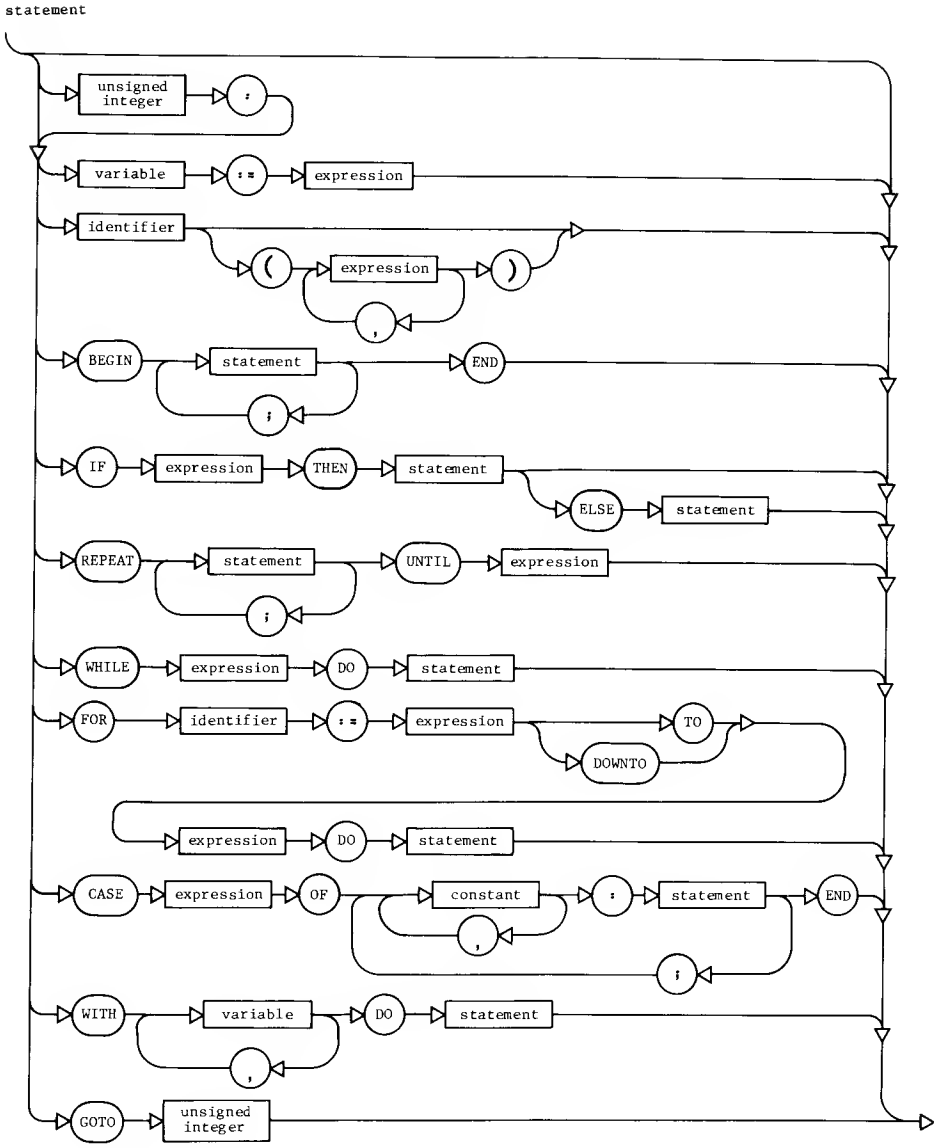
variable



1. If the identifier at the top of the diagram is that of an array, the expression(s) in square brackets may be used to subscript it. The values of the expressions(s) must be compatible with subscript types declared for the array.

2. If the identifier at the top of the diagram is that of a record, it may be followed by a period and a second identifier. The second identifier must be the identifier of one of the fields of the record.

3. If the identifier at the top of the diagram is that of a pointer, it may be followed by the up-arrow character.

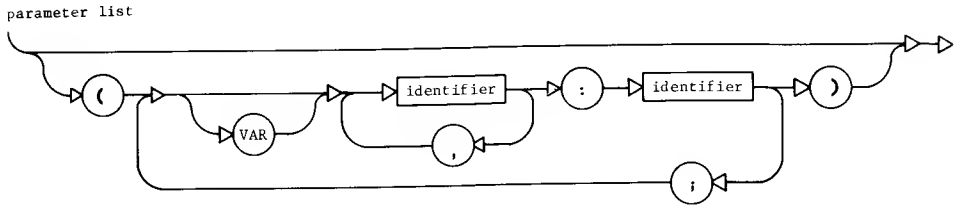


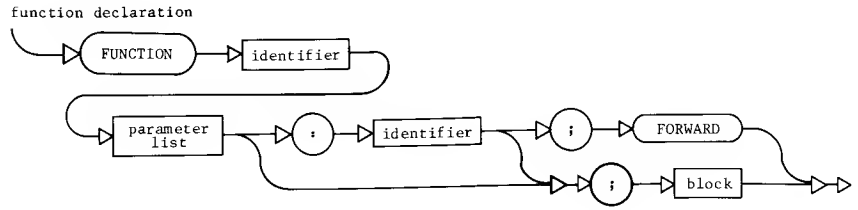
1. Note that there is a "null" path through this diagram, across the top and down the right-hand side without including anything. This represents what happens when a superfluous semicolon occurs in a program.

2. The unsigned integer at the top of the diagram is a label, and must have been declared in a LABEL declaration.

3. The identifier in the third line of the diagram (above BEGIN) must be the identifier of a procedure.

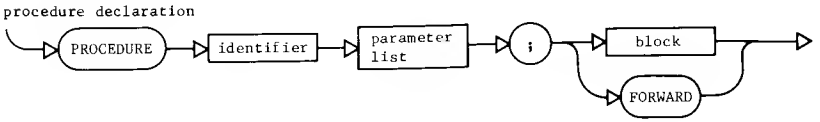
4. The expression in an IF, REPEAT, or WHILE statement must have a BOOLEAN value.





This diagram shows all of the forms a function declaration can take:

- The normal form includes a parameter list (which may be null) and the colon followed by an identifier (which must be that of a type). The declaration ends with a block.
- The FORWARD declaration is like the normal form except that the word FORWARD is used instead of a block.
- Following a FORWARD declaration, the function declaration has no parameter list or type identifier and ends with a block.

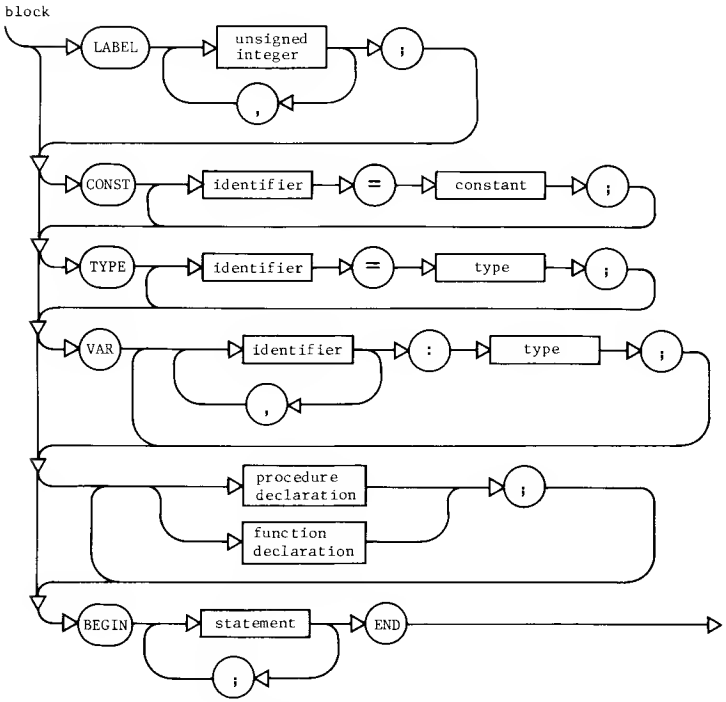


This diagram shows all of the forms a procedure declaration can take:

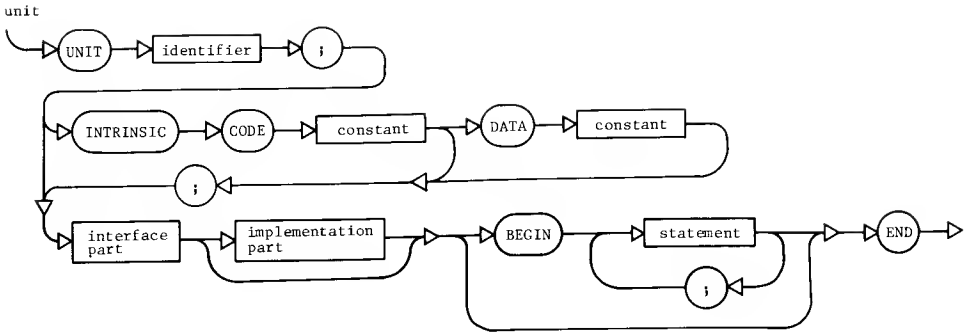
- The normal form includes a parameter list (which may be null). The declaration ends with a block.

- The FORWARD declaration is like the normal form except that the word FORWARD is used instead of a block.

- Following a FORWARD declaration, the procedure declaration has no parameter list and ends with a block.

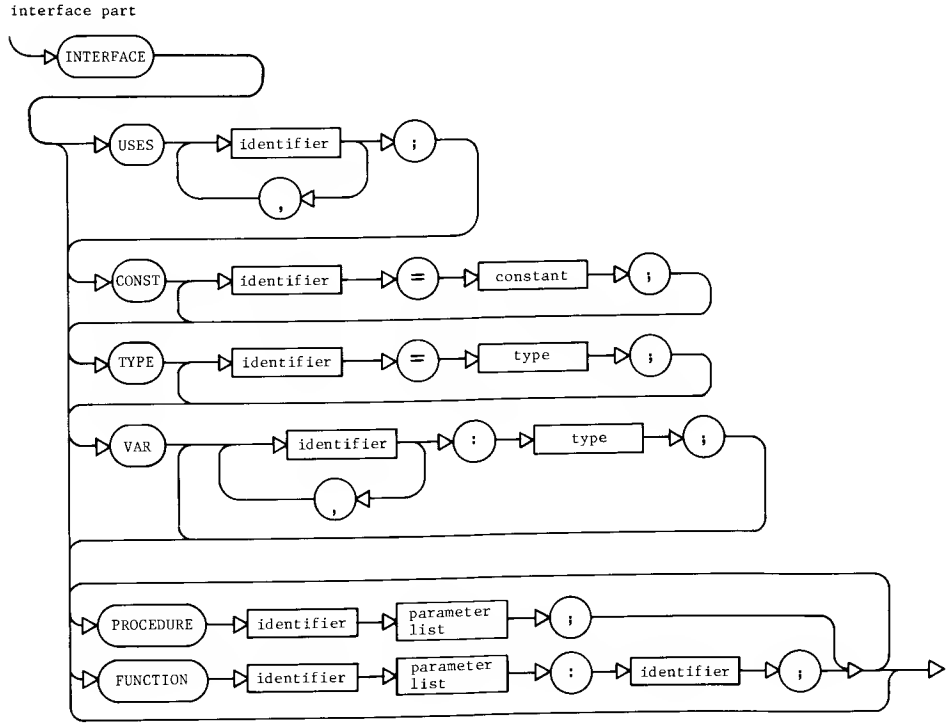


This is one of the fundamental structural units: it contains all the local data declarations (except parameters) and all the statements for one program, procedure, or function.

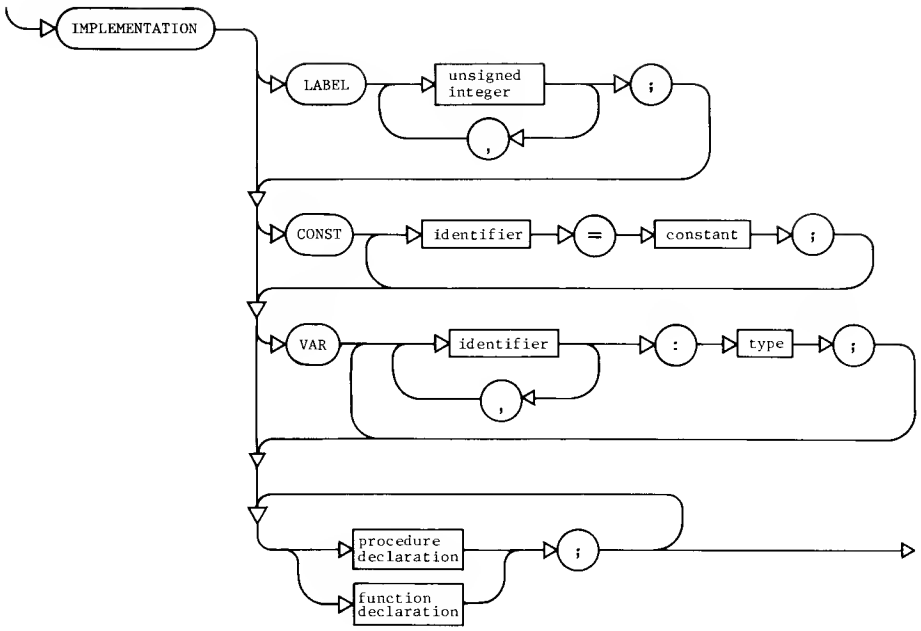


1. In an intrinsic unit, the constants following CODE and DATA must be integers and should be carefully chosen.

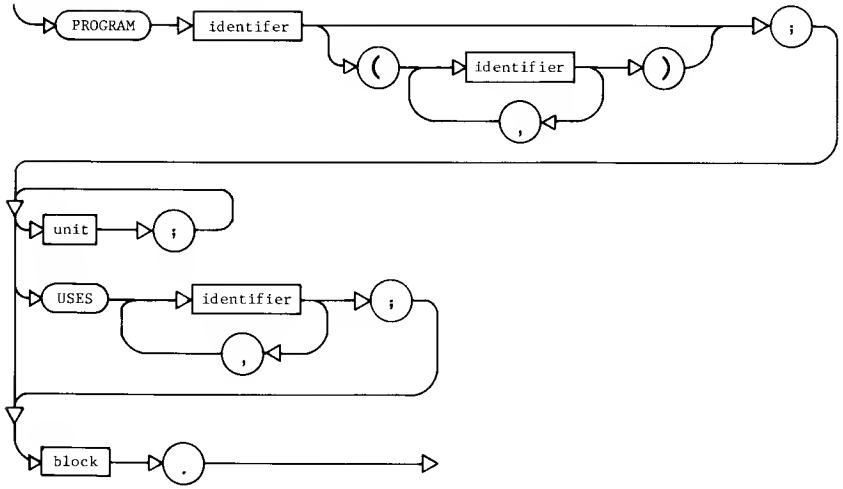
2. The words BEGIN and END with the statements between them are the "initialization" of the unit.



implementation part



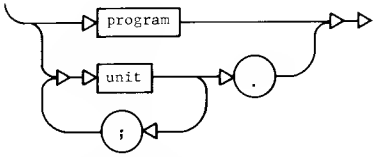
program



1. The program heading may contain identifiers in parentheses in accordance with Standard Pascal syntax. However the identifiers are ignored.

2. Note that any units defined in the program must immediately follow the program heading. This would normally be done only for test purposes.

compilation



A compilation is simply something that the compiler can compile. This may be a program (which may contain units), or one or more units separated with semicolons and ending with a period.

INDEX

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Addendum to the

Apple Pascal

Language Reference Manual

TABLE OF CONTENTS

INTRODUCTION

This document is an addendum to the Apple Pascal Language Reference Manual. Most of the items described are features that have been added to the system since the printing of the manual. Corrections to the manual are also included.

ONE-DRIVE STARTUP

The one-drive startup described on pages 148-15Ø of the Apple Pascal Language Reference Manual is not correct. Instead a one-drive startup works as follows:

Insert the diskette marked APPLE3: in the disk drive. Close the door to the disk drive and turn on the computer. The message

```
APPLE II
```

will appear on the screen and the disk drive's IN USE light will come on. The disk drive emits a whirring sound, lights up for a second or so with a screenful of black at-signs (@) on a white background, and then goes black again. Next the following message is displayed:

```
INSERT BOOT DISK WITH SYSTEM.PASCAL
ON IT. THEN PRESS RESET
```

To complete the booting process, insert APPLEØ: and then press RESET. After about 1Ø seconds, this message appears in the center of the screen:

```
WELCOME APPLEØ, TO APPLE II PASCAL 1.1
BASED ON USCD PASCAL II.1
CURRENT DATE IS 14-AUG-8Ø

(C)  APPLE COMPUTER INC. 1979, 198Ø
(C)  U.C. REGENTS 1979
```

The date may be different. The top of the screen will contain the Command level prompt line.

APPLE Part #Ø31-Ø1Ø1-ØØ

CHAINING PROGRAMS

Version 1.1 provides a new UNIT called CHAINSTUFF in the SYSTEM.LIBRARY file. This unit allows one program to "chain to" another program. This means that the first program specifies the second one by giving its filename; the system then executes the second program as soon as the first one terminates normally.

The CHAINSTUFF unit also allows the first program to pass a STRING value to the second program; note that this allows almost any information to be passed, since the string can be a filename and can thus specify a communications file containing almost anything. CHAINSTUFF also allows a program to turn the system swapping feature on and off. (System swapping is a new feature described in the Addendum to the Apple Pascal Operating System Reference Manual.)

CHAINSTUFF provides these capabilities in the form of five procedures named SETCHAIN, SETCVAL, GETCVAL, SWAPON, and SWAPOFF. To use these procedures, the program must have a USES declaration immediately after the program heading:

```
PROGRAM STARTER;
USES CHAINSTUFF;
...
```

The SYSTEM.LIBRARY file must be on line when the program is compiled and executed.

THE SETCHAIN PROCEDURE

The SETCHAIN procedure call has the form

```
SETCHAIN ( NEXTFILE )
```

where NEXTFILE is a STRING value (up to 23 characters). It should be either the name of a code file, or the name of an exec file with the prefix EXEC/. As soon as the program terminates normally, the system will proceed to execute the file whose name is the value of NEXTFILE.

The file is executed exactly as if the X(ecute command had been used; thus it is not necessary to supply the suffix .CODE for a code file or .TEXT for an exec file.

If the program is halted because of any run-time error, the chaining does not occur. Note that this includes a halt caused by the HALT procedure. However a termination caused by the EXIT procedure is considered a normal termination and the chaining will work.

THE SETCVAL PROCEDURE

The SETCVAL procedure call has the form

```
SETCVAL ( MESSAGE )
```

where MESSAGE is a STRING value (up to 8Ø characters). SETCVAL stores the MESSAGE in a system location called CVAL, where it can be picked up by another program.

THE GETCVAL PROCEDURE

The GETCVAL procedure call has the form

```
GETCVAL ( MESSAGE )
```

where MESSAGE is a STRING variable whose value is altered by GETCVAL. GETCVAL picks up the current value of CVAL from the system and stores it in the MESSAGE variable. Note that if CVAL has not been set by another program (using SETCVAL), then the value of CVAL is a zero-length string. Once CVAL has been set, it remains set to the same STRING value until it is changed or the system is reinitialized or rebooted.

SWAPON AND SWAPOFF

These procedures have no parameters. They allow a program to turn the system swapping feature on or off upon termination of the program (before chaining to another program).

AN EXAMPLE OF CHAINING

Suppose that a diskette named GAMES: contains a collection of game programs whose code files have the following names:

```
CHESS.CODE
CHECKERS.CODE
BLASTOFF.CODE
GOMOKU.CODE
BKGAMMON.CODE
BLACKJCK.CODE
HEARTS.CODE
SPROUTS.CODE
```

The user could use the Filer to display a list of filenames on the GAMES: diskette, then return to the Command level and use X(ecute to execute a selected program. Instead, however, you can write a "front-end" program to display a menu of all the available games; the user chooses one by typing a number, and the front-end program chains to the selected game program:

```
PROGRAM FRONT;
USES CHAINSTUFF;

VAR GAMENUM: INTEGER;

BEGIN
(*Display a greeting*)
  WRITELN('WELCOME TO GAMES!');
  WRITELN;
(*Display the menu*)
  WRITELN('SELECT A GAME FROM THE LIST BY TYPING ITS NUMBER:');
  WRITELN;
  WRITELN('1 -- CHESS');
  WRITELN('2 -- CHECKERS');
  WRITELN('3 -- BLASTOFF');
  WRITELN('4 -- GOMOKU');
  WRITELN('5 -- BACKGAMMON');
  WRITELN('6 -- BLACKJACK');
  WRITELN('7 -- HEARTS');
  WRITELN('8 -- SPROUTS');
  WRITELN;
(*Get a number from the user*)
  WRITE('TYPE A NUMBER FROM 1 THROUGH 8, THEN PRESS RETURN: ');
  READLN(GAMENUM);
(*Make sure the number is valid*)
  WHILE NOT (GAMENUM IN [1..8]) DO BEGIN
    WRITE('NUMBER MUST BE FROM 1 THROUGH 8 -- TRY AGAIN: ');
    READLN(GAMENUM)
  END;
(*Set chaining to filename of selected game*)
  CASE GAMENUM OF
    1:  SETCHAIN('GAMES:CHESS');
    2:  SETCHAIN('GAMES:CHECKERS');
    3:  SETCHAIN('GAMES:BLASTOFF');
    4:  SETCHAIN('GAMES:GOMOKU');
    5:  SETCHAIN('GAMES:BKGAMMON');
    6:  SETCHAIN('GAMES:BLACKJCK');
    7:  SETCHAIN('GAMES:HEARTS');
    8:  SETCHAIN('GAMES:SPROUTS')
  END
END.
```

There are several advantages to this. For one thing, the GAMES: diskette may have many other files besides the actual game programs, and this could be confusing to the user. For another, the FRONT program menu gives full and correct names for the games, since it is not limited to 8-character names; thus it lists BACKGAMMON instead of BKGAMMON.

Many game programs ask the user to type in her name, so it can be used in messages and prompts from the program. You could also have the FRONT program get the user's name and pass it to the selected game program. To do this, the FRONT program can declare a STRING variable, NAME, and then include the following lines either just before or just after the CASE statement:

```
(*Get user's name and store it in CVAL*)
WRITE('TYPE YOUR NAME, PLEASE:  ');
READLN(NAME);
SETCVAL(NAME)
```

Now a game program that uses the user's name can obtain it by having its own STRING variable named (for example) UNAME, and then calling GETCVAL:

```
GETCVAL(UNAME)
```

Incidentally, if the FRONT program's codefile is placed on the boot diskette and given the name SYSTEM.STARTUP, the FRONT program will be run automatically as soon as the system is started.

THE "V" OPTION

When a procedure or function has a VAR parameter of type STRING, the actual parameter in each call to the procedure or function is checked by the Compiler to make sure that its maximum length is not less than the maximum length of the formal parameter. In the previous version of the Compiler, there was no such checking.

This checking is controlled by the V option:

Default value: V+

(*$V+*) Turns checking on.

(*$V-*) Turns checking off.

The U- option also turns this checking feature off. Note that if checking is off and the maximum length of the actual parameter is less than the maximum length of the formal parameter, it is possible for the procedure or function to alter bytes of data that are beyond the end of the actual parameter variable. This does not cause a run-time error, but does cause unpredictable results.

THE "SWAPPING" OPTION

With the S+ option of the Compiler, the extra space available for symbol-table storage is about 53ØØ words (compared to about 39ØØ in the previous version). Going from S+ to S++ gives about 15ØØ words more.

STRINGS

The manual for Version 1.1 states (on page 9) that one string is "greater than" another if it would come first in an alphabetic list of strings. This is backwards: it should state that one string is "less than" another if it would come first in an alphabetic list of strings.

TURTLEGRAPHICS

The system will automatically return to text mode if a program terminates while in graphics mode. This applies to both normal termination and to a halt caused by a run-time error.

Also, the manual erroneously states that the TURTLEGRAPHICS procedures will accept REAL parameters for X,Y coordinates, and convert them to INTEGER values. Actually, X,Y coordinates are parameters of type INTEGER and if a REAL value is supplied an error results.

ERROR MESSAGES

The following new compiler error messages have been added:

175: Actual parameter max string length < var formal max length

4Ø8: (*$S+*) Needed to compile units

MEMORY SPACE FOR COMPILER

When compiling a very large program, it is possible for the compiler to run out of memory space. There are several remedies to try when this happens:

Use the command-level swapping option to get 11ØØ words of additional memory space.

Use the compiler swapping option (*$S+*) or if necessary, (*$S++*).

If the program uses "include" files, use the Filer's M(ake command to create a 4-block file named SYSTEM.SWAPDISK on the same diskette that contains the Compiler. This allows a segment of the compiler to be swapped out onto the diskette before the operating system segment that opens files is swapped in.

With include files, it also helps to have the (*$I filename*) option in the declaration section of a procedure (before the BEGIN of the

procedure body). This only helps if the compiler swapping option is on and there is a SYSTEM.SWAPDISK file. In the declaration section of a procedure is where the largest section of the compiler can be swapped out to make room for the operating system segment that opens files.

FILE SPACE FOR COMPILER

In Version 1.1, when the code file is automatically sent to the workfile the default size for the file is [*]. In all other cases, the default size is [Ø], which means that the code file will be allocated all of the largest space available on the diskette that it is sent to. If there is only one available space on the diskette, the code file takes all of it.

This can cause problems if the code file is on the same diskette used for the listing file. The Compiler will fail if it tries to create a listing file and the code file has taken all the available space on the specified diskette.

If you run into these problems, specify a different diskette for the code file or the listing file, or specify a definite length for the code file that will leave enough room for other required files.

PROGRAM SEGMENTATION

The information in this section is not needed for simple programs, but can be crucial for programs that are large or complex. This section supplements the information in Chapters 4 and 5 of the Apple Pascal Language Reference Manual. It also describes a new Compiler option, the "Next Segment" option, which is not described in the manual.

To make the most efficient use of the memory space available for program code and data, programs can be divided into segments. This section gives essential information on how the Pascal System implements segmentation.

SEGMENTS

A segment is code that can be loaded into memory and executed, without any other code necessarily being in memory at the same time. Every program consists of at least one segment, and some programs consist of many segments. Whenever a program is compiled, the Compiler and Linker create the following segments in the code file:

- Each SEGMENT procedure or function becomes a segment in the code file.
- Each Regular UNIT that the program uses becomes a segment in the code file.
- The program itself becomes a segment in the code file. This includes the program's non-SEGMENT procedures and functions.

Similarly, whenever a Regular UNIT is compiled, the result is a code segment for the UNIT itself, plus an additional segment for each Regular UNIT that is used within the UNIT being compiled. (Note that SEGMENT procedures and functions are not allowed inside UNITs.)

When an Intrinsic UNIT is compiled, it produces a code segment, and may produce a data segment as well. (Note that an Intrinsic UNIT cannot USE a Regular UNIT.)

Note that segments do not nest -- every segment is just one segment and does not contain any other segments. For example, if a SEGMENT procedure contains another SEGMENT procedure, the result is two distinct code segments.

THE SEGMENT DICTIONARY

Every code file (including library files) contains information called a segment dictionary. This contains an entry for each segment in the code file; the entry has all the information the system needs to load and execute the segment.

The segment dictionary has slots for 16 entries. Therefore, one code file can contain at most 16 segments. In the case of a program, this implies one segment for the program itself, one for each SEGMENT procedure or function, and one for each Regular UNIT used by the program.

Note that Intrinsic UNITs used by a program do not require entries in the segment dictionary of the program's code file. This is because an Intrinsic UNIT's code segment is never in the program's code file -- it is in a library file, and appears in the library file's segment dictionary.

Therefore a program can have a maximum of 16 segments, not counting segments from Intrinsic UNITs. Counting segments from Intrinsic UNITs, a program can have up to 26 segments as explained below.

THE RUN-TIME SEGMENT TABLE

When a program is executed, the Pascal Interpreter uses a segment table which contains an entry for each segment that is used in executing the program. This table thus contains the following entries:

- Entries for 6 segments that the system uses when executing a user program
- An entry for each segment in the segment dictionary of the program's code file
- An entry for each Intrinsic UNIT segment (both data and code segments).

The segment table has slots for up to 32 entries. Since the system uses 6, this means that a program can have up to 26 segments altogether. Remember that only 16 can be in the program's code file; any excess over 16 must be Intrinsic UNIT segments.

SEGMENT NUMBERS

Every segment has a segment number in the range Ø..31. At run time, no two segments in the segment table can have the same number, since the numbers are used to index the table. A segment number is assigned to a program segment when the segment's entry is placed in the code file's segment dictionary (before run time). Numbers are assigned as follows:

- The program itself is Segment 1.
- The segments used by the system are Ø and 2..6.
- The segment number of an Intrinsic UNIT segment is determined by the UNIT's heading, when the Intrinsic UNIT is compiled. (These numbers can be found by examining the segment dictionary of the SYSTEM.LIBRARY file with the LIBMAP utility program.)
- The segment numbers of Regular UNIT segments and of SEGMENT procedures and functions are automatically assigned by the system; they begin at 7 and ascend. Note that after a Regular UNIT is linked into a program, it may not have the same segment number shown for it in the library's segment dictionary when the library is examined with LIBMAP.

To summarize the above, the segment numbers of the program itself, the segments used by the system, and any Intrinsic UNITs used by the program are fixed before the program is compiled; the segment numbers of Regular UNITs and of SEGMENT procedures and functions are not fixed, and are assigned as the program is compiled and linked, in ascending sequence beginning with 7.

Normally, the only time you need to specify segment numbers is in writing an Intrinsic UNIT. You should choose segment numbers that will not conflict with any of the fixed numbers Ø..6 or with any other Intrinsic UNIT that might be used in the same program as the UNIT you are writing.

Intrinsic UNIT segment numbers should also avoid conflict with numbers that might be assigned automatically to Regular UNITs and SEGMENT procedures. However, when unavoidable conflicts arise there is a solution: the new version of the Compiler has a "Next Segment" option which can specify the next automatically assigned segment number. This is explained below.

THE "NEXT SEGMENT" OPTION

This is a new Compiler option which allows you to specify the segment number of the next Regular UNIT, SEGMENT procedure, or SEGMENT function encountered by the Compiler. By default, the segment number is assigned automatically as described above.

(*$NS num*) Sets the next segment number to num, where num is an integer in the range 1..3Ø.

The NS option is ignored if it precedes the program heading; this means that it cannot be used to specify the segment number of the program itself.

The NS option will only work if the specified number is greater than the "default" number that would be automatically assigned. If the number specified in the NS option is less than or equal to the default segment number, the option is ignored. Also, the NS option will not work if the specified number is greater than 3Ø (segment number 31 can only be used for an Intrinsic UNIT).

For example, suppose that you want to use an Intrinsic UNIT named ZEBRA, whose code segment number is 7 and whose data segment number is 8. (Normally, such numbers should be avoided in writing Intrinsic UNITs.) Your program also contains a SEGMENT procedure:

```
PROGRAM ELEPHANT;
USES ZEBRA;
...
SEGMENT PROCEDURE HORSE;
...
```

The Compiler will automatically compile the HORSE procedure as segment number 7, and when you try to execute the program the Interpreter will halt with an error message because the program has two different segments with the number 7. There are two remedies: recompile ZEBRA with different segment numbers (if you have the source for ZEBRA) or use the NS option in your program:

```
PROGRAM ELEPHANT;
USES ZEBRA;
...
(*$NS 9*)
SEGMENT PROCEDURE HORSE;
...
```

Now HORSE will become segment 9 instead of segment 7, and the conflict is avoided.

LOADING OF SEGMENT PROCEDURES AND FUNCTIONS

Normally, the code of a SEGMENT procedure or function is in memory only while it is active; that is, it is loaded from diskette each time the procedure or function is called, and unloaded as soon as it finishes executing. The following program illustrates this:

```
PROGRAM ONE; (*Segment ONE is always in memory.*)

  SEGMENT PROCEDURE ALPHA; (*In memory only when active.*)
    BEGIN
      ...
    END;

  SEGMENT PROCEDURE BRAVO; (*In memory only when active.*)
    SEGMENT PROCEDURE CHARLIE; (*In memory only when active.*)
      BEGIN (*Body of CHARLIE*)
        ...
        ALPHA; (*When this is executed, the segments in
                 memory are ONE, ALPHA, BRAVO, and CHARLIE.*)
        ...
      END;
    BEGIN (*Body of BRAVO*)
      ...
      CHARLIE; (*When this starts executing, the segments in
                 memory are ONE, BRAVO, and CHARLIE.*)
      ALPHA; (*When this is executed, the segments in
               memory are ONE, BRAVO, and ALPHA.*)
      ...
    END;

  BEGIN (*Body of ONE*)
    ...
    ALPHA; (*When this is executed, the segments in
             memory are ONE and ALPHA.*)
    BRAVO; (*When this starts executing, the segments in
             memory are ONE and BRAVO.*)
    ...
  END.
```

The "Resident" option can be used to alter this, as explained below.

LOADING OF UNIT SEGMENTS

Normally, all segments of UNITs used by a program are loaded automatically before the program begins executing, and remain in memory throughout program execution. For example, consider the following program where DELTA and GAMMA are two UNITs, either Regular or Intrinsic:

```
PROGRAM TWO
USES DELTA, GAMMA;
...
BEGIN
  ...
END.
```

Throughout program execution, the segments in memory are TWO, DELTA, and GAMMA. This can be altered by the "Noload" option, as explained below.

THE "NOLOAD" OPTION

The "Noload" option, (*$N+*), is described in Chapter 4 of the Apple Pascal Language Reference Manual; this section explains its usage.

The (*$N+*) option is placed at the beginning of the main program body (after the BEGIN). It causes all UNIT segments to be handled in the same way as SEGMENT procedures, during program execution. With this option a UNIT segment is in memory only when something in its INTERFACE part is referenced by the program.

The (*$N+*) option does not prevent the initialization part of a UNIT from being loaded and executed before program execution; but after initialization the UNIT segment is unloaded until it is activated.

Consider the following program, where HUGEPROC is a large procedure and BIGUNIT is a large UNIT. The system does not have enough memory to hold HUGEPROC and BIGUNIT at the same time, along with the program itself.

```
PROGRAM THREE;
USES BIGUNIT;

SEGMENT PROCEDURE HUGEPROC;
  BEGIN
    ...
  END;

BEGIN
  (*$N+*) (*Keeps BIGUNIT out of memory until needed.*)
  HUGEPROC;
  ...
  CALCULATE; (*A procedure in BIGUNIT*)
  ...
  HUGEPROC
END.
```

First HUGEPROC is called; BIGUNIT is not in memory because of the (*$N+*) option. When CALCULATE is called, HUGEPROC is not in memory since it is a SEGMENT procedure. As soon as no part of BIGUNIT is active, it is again swapped out of memory, and HUGEPROC can be called again.

THE "RESIDENT" OPTION

The "Resident" option is described in Chapter 4 of the Apple Pascal Language Reference Manual; this section explains its usage.

The "Resident" option is placed at the beginning of the body of a procedure or function (after the BEGIN). It alters the handling of segments that would otherwise be in memory only when active: that is, SEGMENT procedures and functions, and UNITs under the "Noload" option. When such a segment is named in the "Resident" option, it is immediately loaded into memory and remains there as long as the procedure or function containing the "Resident" option is active. For example, consider the following program:

```
PROGRAM FOUR;
USES BIGUNIT;

SEGMENT PROCEDURE HUGEPROC;
  BEGIN
    ...
  END;

PROCEDURE CALLHUGEPROC;
  VAR I: INTEGER;
  BEGIN
    FOR I:=1 TO 100 DO HUGEPROC
  END;

PROCEDURE CALLCALCULATE;
  VAR I: INTEGER;
  BEGIN
    FOR I:=1 TO 100 DO CALCULATE (*A procedure in BIGUNIT*)
  END;

BEGIN
  (*$N+*) (*Keeps BIGUNIT out of memory until needed.*)
  HUGEPROC;
  ...
  CALCULATE;
  ...
  CALLHUGEPROC;
  ...
  CALLCALCULATE
END.
```

This resembles the previous example, but the CALLHUGEPROC and CALLCALCULATE procedures are new. As written, these two procedures have a problem: since HUGEPROC is a SEGMENT procedure, it will be swapped in from diskette 1ØØ times when CALLHUGEPROC executes, and because of the (*$N+*) option in the main program body, BIGUNIT will be swapped in 1ØØ times when CALLCALCULATE executes. This is obviously undesirable, and it can be prevented by using the "Resident" option in each of these procedures:

```
PROCEDURE CALLHUGEPROC;
  VAR I: INTEGER;
  BEGIN
  (*$R HUGEPROC*)
    FOR I:=1 TO 1ØØ DO HUGEPROC
  END;

PROCEDURE CALLCALCULATE;
  VAR I: INTEGER;
  BEGIN
  (*$R BIGUNIT*)
    FOR I:=1 TO 1ØØ DO CALCULATE (*A procedure in BIGUNIT*)
  END;
```

Now HUGEPROC will be kept in memory as long as CALLHUGEPROC is active, and BIGUNIT will be kept in memory as long as CALLCALCULATE is active.

Finally, note that the "Resident" option can be applied to more than one segment, by separating the names of segments with commas as in the following example:

```
(*$R ALPHA,BETA,GAMMA*)
```

where ALPHA, BETA, and GAMMA are names of segments (UNITs, SEGMENT procedures, or SEGMENT functions). The option shown would make all three segments resident in the procedure containing the option.